Catarina da Costa Boucinha

Operational Semantics for Linear Languages

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012

Catarina da Costa Boucinha

Operational Semantics for Linear Languages

Thesis submited to the Faculdade de Ciências of
Universidade do Porto to obtain Masters degree
in Computer Science

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012

To my parents and all my friends...

Acknowledgments

First of all I would like to thank my supervisor Sandra Alves for all her help throughout the writing of this thesis, especially bearing with my constant nervousness and always helping to calm me down. This thesis would have never been possible without her help and support. I would also like to thank Mário Florido for his help, always with helpful suggestions and advice.

I would like to thank all my friends, especially Eva Maia, for always getting me to go further than what I though I could go, for all the advice, knowledge and friendship shared throughout the years, and of course for all the coffee breaks. And to Marcela Oliveira, for all the support and her caring shoulder, for being my family when I lived in Porto and for putting up with my stressed person every single day. I thank all my friends from Ovar, who made my weekends a thing to look forward to and who always listen to my computer "geek" talk. Diana, Luís, Marilia, Nuno and Sofia my wild side thanks you with all its heart. I would also thank my friends who made every train, subway or bus ride an adventure, an epic odyssey of stories that I will remember forever.

And since I would have a very long list of names if I were to mention you all, I would like to thank all my friends that shared with me the adventures that I like to call "the story that I will tell my grandchildren", the sad and happy moments. I would like to thank the ones who shared their knowledge, how to be a better person, how the world works; the ones that sang with me, that grew with me, that traveled the world with me; and the ones who stood on a stage with me and that showed me that I had more courage than what I though.

Last but never least, I would like to thank my family, specially my mother and father, without whom I would have never made it this far. For their constant support through the good and bad decisions that I made, for always working so hard to give me the opportunity to reach this far in the academic world, for their constant advice (even when I did not want to ear it) that helped me be the person I am today, and finally for listening to my never ending constant talking with all the patience in the world.

Abstract

The goal of this thesis is to study the operational semantics of the linear language, System $\mathcal{L}_{rec}$, in order to explore its use as an intermediate language. System $\mathcal{L}_{rec}$ is a linear λ-calculus extended with numbers, pairs and an unbound linear recursor, with a close-reduction strategy. It is compatible with the notion of linear function with a minimal extension to the linear λ-calculus, keeping the system Turing-complete.
The contributions of this thesis are:

- Implementations of interpreters call-by-name for the **PCF** language, System $\mathcal{L}_{rec}$, and its extension with built-in naturals, $\mathcal{L}^{N}_{rec}$ and an interpreter call-by-value for System $\mathcal{L}_{rec}$;
- Implementations of call-by-name abstract machines for the **PCF** language, the System $\mathcal{L}_{rec}$ and $\mathcal{L}^{\mathbf{nat}}_{rec}$;
- A compilation from **PCF** to System $\mathcal{L}_{rec}$ and $\mathcal{L}^{\mathbf{nat}}_{rec}$;
- Several benchmarks on the number of reductions for its different machines.

Resumo

Esta tese tem como objectivo estudar as semânticas operacionais de uma linguagem linear, Sistema $\mathcal{L}_{rec}$, de maneira a explorar o seu uso como uma linguagem intermédia. O Sistema $\mathcal{L}_{rec}$ é um λ-calculus extendido com números, pares e um recursor linear não ligado, com uma estratégia de redução fechada. É compativel com a noção de função linear com uma extensão mínima ao λ-calculus linear, mantendo o sistema Turing completo.
As contribuições desta tese são:

- Implementações de interpretadores de chamada-por-nome para a linguagem **PCF**, Sistema $\mathcal{L}_{rec}$ e uma extensão deste com naturais *built-in*, $\mathcal{L}^{\mathbf{nat}}_{rec}$, e um interpretador de chamada-por-valor para o Sistema $\mathcal{L}_{rec}$;
- Implementação de máquinas abstractas de chamada-por-nome para a linguagem **PCF**, para o Sistema $\mathcal{L}_{rec}$ e $\mathcal{L}^{\mathbf{nat}}_{rec}$;
- Uma compilação de **PCF** para o Sistema $\mathcal{L}_{rec}$ e $\mathcal{L}^{\mathbf{nat}}_{rec}$;
- Várias conclusões sobre o número de reduções para as diferentes máquinas.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of this thesis is to study operational semantics for the linear language, $\mathcal{L}_{rec}$, and to compare it with standard operational semantics for functional languages based on λ-calculus.
System $\mathcal{L}_{rec}$ [Alves 11] is defined as a linear λ-calculus extended with numbers, pairs and an unbounded linear recursor, which gives a Turing-complete system. It is used, in this thesis, as a base for creating an operational semantics for linear languages. We use System $\mathcal{L}_{rec}$ because we believe it will show better results as an intermediate code than the other linear languages due to its unbounded recursor.
We will compare it with **PCF**, an also Turing-complete programming language with ground types, roughly close to λ-calculus with a fixed point operator, which makes it the ideal programming language to compare with System $\mathcal{L}_{rec}$.
The **PCF** language (Programming Language for Computable Functions) was introduced by Plotkin [Plotkin 77], as a programming language for computational functions based on LCF [Scott 93]. It is a variant of the typed λ-calculus with the addition of ground types **int** and **bool** and a restrict relation of conversion.
In order to obtain a Turing-complete system, **PCF** extends the simply typed λ-calculus with a fixed point operator. However, it is based on the existence of a non-linear condition, which discards the possibility of infinite computation on the branches. On the other hand, System $\mathcal{L}_{rec}$ uses an unbound recursor with a built-in test over pairs that allows the encoding of finite iterations and minimization, obtaining a Turing-complete linear λ-calculus.
Starting from these two languages, the contributions of this thesis are:

- An implementation of call-by-name interpreter for the **PCF** language, and also

an abstract machine based on the Krivine machine built with closures and an environment;

- An implementation of call-by-name and call-by-value interpreters for the System $\mathcal{L}_{rec}$;
- An abstract machine for System $\mathcal{L}_{rec}$. Because $\mathcal{L}_{rec}$ is a syntactically linear calculi (each variable is linear on the terms) there is no need for an environment, which differentiates it from the abstract machines for non-linear λ-calculus. It is also given a variant of the abstract machine for System $\mathcal{L}_{rec}$ with built-in naturals, $\mathcal{L}^{\mathbf{nat}}_{rec}$;
- A compilation from **PCF** to $\mathcal{L}_{rec}$ and $\mathcal{L}^{\mathbf{nat}}_{rec}$, and its implementation, so we can explore the application of the system as an intermediate language;
- To conclude this thesis, a comparison between the results of the evaluations of the languages expressions was made using an abstract time measure cost inspired by [Lago 06], which gives us an invariant cost measure based on the number of reductions of an expression that takes into consideration the size of the expression.

1.1 Motivation and Related Work

Syntatically linear languages have several implementation advantages: knowing that a function is linear is a property that a compiler can take advantage to optimize code, as well as in program analysis.
The linear λ-calculus is a subset of the λ-calculus in which no variable occurs more that once in any subterm of any term, and for any function the argument occurs in the body of the function. Although it has very nice computational properties, it has a very limited computational power: every term reduces in time linear to its size.
There are several works that explore linearity in the λ-calculus, many of those derived from Girard's Linear Logic [Girard 87] (a logic where hypothesis are looked as resources, which are consumed by proofs, contrary to classical logic where hypothesis can be used as many times as needed). In particular an extension of the linear λ-calculus with bounded iterator of closed functions proved to capture exactly the class of primitive recursive functions [Lago 09]. In [Alves 10] another extension of the linear λ-calculus with bounded iteration was presented, that imposed a closed-on-reduction condition on iterated functions which proved to be computationally equivalent to

Gödel's System T [Girard 89] (Gödel's System T is a simply typed λ-calculus with numbers and booleans, a condition and a bounded recursor).
We point out to [Asperti 98, Girard 98, Asperti 02, Lafont 04, Terui 01, Baillot 04] for the definition of several other linear calculi for capturing specific complexity classes. There is also previous work that uses linear types to characterize computations with time bounds [Hofmann 99].

1.2 Overview of the Thesis

The rest of the thesis is organized in the following way:

Chapter 2: Background In this chapter, a background is given in λ-calculus, Simple Typed λ-calculus and Operational Semantics, explaining its basic concepts and notions. Also, a brief explanation of the Krivine abstract machine for the λ-calculus, which uses a call-by-name evaluation.

Chapter 3: PCF Language Here we give the syntax and type system for the **PCF** language, an abstract stack-machine for a call-by-name evaluation of the language, and we explain the implementation of the evaluation and of the machine.

Chapter 4: System $\mathcal{L}_{rec}$ A syntax and type system for System $\mathcal{L}_{rec}$ is given in Chapter 4 along with an abstract stack-machine, also with a call-by-name evaluation. In the end of this chapter, an implementation of System $\mathcal{L}_{rec}$ is also given.

Chapter 5: Compiling In this chapter we show the encoding from **PCF** to System $\mathcal{L}_{rec}$, its implementation and the comparison of the results obtained. Also, a variant of System $\mathcal{L}_{rec}$ with built-in naturals is given, with its implementation and a comparison the results obtained previously against this new implementation.

Chapter 6: Conclusions In this final chapter we state the conclusions of this work along with the outline of future work.

Chapter 2

Background

2.1 Lambda-Calculus

In this section we briefly describe the λ-calculus, defined by Alonzo Church [Church 32] in 1932, which is a Turing-complete computational model that has a very simple syntax (variables, function abstraction and function application) and a main rewrite rule (β-reduction), based on the notion of substitution.

2.1.1 Syntax

The λ-calculus models the definition and application of functions, based on the notion of substitution. The application (MN) represents the application of the function represented by M to the argument represented by N. The abstraction $(\lambda x.M)$ represents the function f such that $f(x) = M$. The application of this function f to N results in the substitution of x for N in M, i.e. $f(N)$.

Definition 2.1.1. ***(λ-calculus)*** *Let $\mathbb{V}$ be an infinite set of variables. The set of λ-terms Λ, is build from the set $\mathbb{V}$ using application and abstraction in the following way:*

$$\begin{array}{lrcl} & x \in \mathbb{V} & \Rightarrow & x \in \Lambda \\ \textit{(Application)} & M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda \\ \textit{(Abstraction)} & M \in \Lambda, x \in \mathbb{V} & \Rightarrow & (\lambda x.M) \in \Lambda \end{array}$$

Notation: We use the symbol $\equiv$ to the denote syntactic equality between terms.

Since we consider that the application is left associative, and the abstraction is right associative, we will use the following abbreviations:

- $(M_1 M_2 \dots M_n) \equiv (\dots (M_1 M_2) \dots M_n)$
- $(\lambda x_1 x_2 \dots x_n.M) \equiv (\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots)))$

Which means that the term $(\lambda x.(\lambda z.((xz)(\lambda y.(yx)))))$ can be written as $(\lambda xz.xz(\lambda y.yx))$.

2.1.2 Variables and Substitution

We will distinguish the type of variable occurrences, formalizing the concepts of *free* and *bound* variables of a term.

Definition 2.1.2. ***(Free Variables)*** *Let $M \in \Lambda$, the set $FV(M)$ of free variables of M is inductively defined as follows:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

Definition 2.1.3. ***(Bound Variables)*** *Let $M \in \Lambda$, the set $BV(M)$ of bound variables of M is inductively defined as follows:*

$$\begin{aligned} BV(x) &= \emptyset \\ BV(MN) &= BV(M) \cup BV(N) \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \end{aligned}$$

A λ-term is closed if and only if $FV(M) = \emptyset$.
Note that the sets of free and bound variables of a term are not necessarily disjoint, for example, x occurs both *free* and *bound* in the term $x(\lambda xy.x)$.

Definition 2.1.4. ***(Substitution)*** *The result of substituting the free occurrences of x by L in M (denoted by $M[L/x]$) is defined as:*

$$x[L/y] \equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases}$$

$$(MN)[L/y] \equiv (M[L/y])(N[L/y])$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} (\lambda x.M) & \text{if } x \equiv y \\ (\lambda x.M[L/y]) & \text{otherwise} \end{cases}$$

One problem that can arise from variable substitution is the *capture* of free variables. As an example, consider the λ-term $(\lambda xy.x)$. We would expect to obtain N when applied to two arguments N and P. However, if $N \equiv y$ then we would get $(\lambda xy.x)[N/x][P/y] \equiv P$, because the free occurrence of x becomes a bound occurrence of y. To avoid this problem, the substitution $M[N/x]$ should only be permitted if x does not occur free in any subterm of M of the form $\lambda y.P$, and $y \in FV(N)$, in which case we say that x is substitutable by N in M.
This condition is ensured if the set of bound variables of M is disjoint from the set of free variables of N:

$$BV(M) \cap FV(N) = \emptyset$$

We can always rename the bound variables in M to ensure that this condition is guarantied. This change in the bound variables is called α-conversion.

Definition 2.1.5. ***(α-conversion)*** *A change of a bound variable x in a term M is the substitution of all subterms of M of the form $(\lambda x.N)$ by $\lambda y.(N[y/x])$, where y does not occur in N.*

This change of bound variables does not alter the function, in fact, it preserves the meaning of the term. This notion is called *α-congruence*.

Definition 2.1.6. ***(α-congruence)*** *The terms M and N are α-congruent, (notation $M \equiv_\alpha N$), if N can be obtained from M, by a series of changes of bound variables, and vice-versa.*

For example:

$$\lambda x.xy \equiv_\alpha \lambda z.zy \not\equiv_\alpha \lambda y.yy.$$

Assume from now on, that the sets of free and bound variables are always disjoint, which is known as the Barendregt's name convention [Barendregt 97]. Therefore any substitution $M[N/x]$ is valid. Also, do not differentiate terms that are α-congruent (for instance $\lambda x.x \equiv \lambda y.y$).

2.1.3 The β-reduction

We will now describe a reduction relation on Λ [Barendregt 84], that together with Λ, defines the reduction system for which some properties will be discussed.

Definition 2.1.7. ***(β-reduction)*** *The following contraction rule defines the notion of β-reduction on Λ:*

$$\beta : (\lambda x.M)N \to M[N/x], \quad M, N \in \Lambda$$

A λ-term of the form $(\lambda x.M)N$ is called a *β-redex* and $M[N/x]$ is its *β-contractum*. A λ-term M is reduced to N in one β-reduction step, and is written as $M \to_\beta^1 N$, if N can be obtained by substituting on M a *β-redex* by its *β-contractum*. We write $M \to_\beta^* N$, for the reflexive and transitive closure of $\to_\beta^*$.

Definition 2.1.8. ***(Normal Form)*** *A term that does not contain any β-redex, does not admit any β-reductions, and is therefore said to be in* β-nf *or just* normal form.

For example, the term $(\lambda x.z)((\lambda xy.xy)(\lambda x.x))$ is not in *β-nf* form since it has a *β-redex* $(\lambda xy.xy)(\lambda x.x)$. The term admits a *$\beta$-nf*, which is z.
The β-reduction system is Church-Rosser [Church 36, Barendregt 84], which ensures that reduction is confluent.

Theorem 2.1.1. ***(Church-Rosser)*** *Let M be a λ-term, if $M \to_\beta^* N_1$ and $M \to_\beta^* N_2$ then, there is a N such that $N_1 \to_\beta^* N$ and $N_2 \to_\beta^* N$.*

A λ-term that has a normal form, which admits an infinite reductions sequence, reaches the normal form if all the subterms of M that do not have normal forms, are erased.

2.1.4 Reduction Strategies

The goal of this section is to define and clarify the relation between programming language concepts such as call-by-name and call-by-value, and λ-calculus concepts such as Normal Order Reduction and Applicative Order Reduction.
We will start by defining a reduction strategy, which is a procedure that finds the next redex to be contracted. That redex can be:

- *Leftmost-Outermost*: the leftmost redex not contained in any other redex;

Reduce Args	**Reduce under Abstractions**	
	Yes	No
Yes	Normal form $E ::= \lambda x.E \mid xE_1 \dots E_n$	Weak Normal Form $E ::= \lambda x.M \mid xE_1 \dots E_n$
No	Head Normal form $E ::= \lambda x.E \mid xM_1 \dots M_n$	Weak Head Normal Form $E ::= \lambda x.M \mid xM_1 \dots M_n$

Table 2.1: Normal Forms, where $M_i \in \Lambda$ (Definition 2.1.1)

- *Leftmost-Innermost*: the leftmost redex that does not contain a redex.

The are other redexes, e.g. *Rightmost-Innermost*, *Rightmost-Outermost*, etc., but in the interest of this thesis, they will not be defined.
Note that a redex is to the left of any other redex if his λ-abstractor is further to the left in the syntactic form of the term. Also note that, in a reduction of a *Leftmost-Outermost* redex, if in an application $(M_1\ M_2)$, if $M_1 \rightarrow_\beta^1 (\lambda x.M)$ then the term $((\lambda x.M)M_2)$ has to be reduced before any redex in M (otherwise it would not be *Outermost*).
Although the definition for Normal Form has already been introduced (Definition 2.1.8), in order to really comprehend the strategies we are going to present, we will have to explore other concepts of normal forms such as *Weak Normal Form*, *Head Normal Form* and *Weak Head Normal Form* (Table 2.1, where E is the term in the relevant normal form). We will distinguish between strategies that choose the next redex to be contracted and strategies that prevent some redex from being contracted, since the set of normal forms will differ depending on the strategy used. Note that not all the strategies reduce under abstractions as it is usual in Functional Programming Languages. The main difference between the strategies relies on whether we reduce under abstractions and whether we reduce the arguments before substitution (in strict languages) or not (in non-strict languages).
We will consider four different reduction strategies:

1. **Call-by-Name Reduction**: consists on reducing the *Leftmost-Outermost* redex that is not under any λ-abstraction, because this strategy does not reduce under abstractions. It reduces to a term in *Weak Head Normal Form*, if the term has one;

2. **Normal Order Reduction**: also consists on reducing the *Leftmost-Outermost*

redex, with the difference that this strategy reduces under λ-abstractions. It reaches to a term in *Normal Form*, if the term has one;

3. **Call-by-Value Reduction**: consists on reducing the *Leftmost-Innermost* redex that is not under any λ-abstraction, since this strategy, as the first one, does not reduce under abstractions. It reduces to a term in *Weak Normal Form*, if the term has one;

4. **Applicative Reduction**: also consists on reducing the *Leftmost-Innermost* redex but as it was for the second strategy, it reduces under λ-abstraction, being this what differentiates it from the previous one. This strategy reaches a term in *Head Normal Form*, if the term has one.

Note that the **Normal Order Reduction**, has the distinct property of being normalizing, in the sense that it reaches the β-normal form if the term has one.

Definition 2.1.9 (Strong Normalization)**.** *If for any λ-term M, all the reduction strategies find its normal form, then M is strongly normalisable.*

A **call-by-name** evaluation is going to be presented for the **PCF** language (Chapter 3) and System $\mathcal{L}_{rec}$ (Chapter 4), and a **call-by-value** evaluation is also going to be presented for System $\mathcal{L}_{rec}$.
For further knowledge on this subject one can read [Sestoft 02].

2.1.5 Linear Calculi

Several subsystems of the λ-calculus can be obtained by restricting the set of terms. Some of those subsystems are the $\lambda_{\mathcal{I}}$-calculus, where function parameters have to occur in the body of the function, the *affine* λ-calculus, where function parameters occur at most one time in the body of the function, and, the one we are going to present in detail, the *linear* λ-calculus.
In the linear λ-calculus, every variable in every term M occurs free exactly once in any subterm of M.

Definition 2.1.10. *Let $\mathbb{V}$ be an infinite set of variables. The set of linear λ-terms, $\Lambda_{\mathcal{L}}$ is inductively defined from $\mathbb{V}$ in the following way:*

$$\begin{array}{rcll} x \in \mathbb{V} & \Rightarrow & x \in \Lambda_{\mathcal{L}} & \\ M, N \in \Lambda_{\mathcal{L}}, FV(M) \cap FV(N) = \emptyset & \Rightarrow & (M\ N) \in \Lambda_{\mathcal{L}} & \textit{(Application)} \\ M \in \Lambda_{\mathcal{L}}, x \in fv(M) & \Rightarrow & (\lambda x.M) \in \Lambda_{\mathcal{L}} & \textit{(Abstraction)} \end{array}$$

All the notions defined for Λ, are defined in an analogous way for $\Lambda_{\mathcal{L}}$. In Chapter 4, we will present a system which extends the linear λ-calculus with natural numbers, pairs and a recursor.

2.2 The Simple Typed Lambda Calculus

In the previous section we discussed the type-free λ-calculus. In this section we will present a simple type system for λ-calculus, the Curry Type System [Curry 34], which initial motivation was to avoid paradoxical uses of the untyped calculus [Church 40]. This system was first studied for the theory of combinators, and was then modified for the λ-calculus in [Curry 58]. The definitions and proofs of results in this section can be found in [Barendregt 84].
In this thesis we will focus on typed calculi, such as **PCF** and System $\mathcal{L}_{rec}$, which both extent the simply typed λ-calculus that we are going to present in this section. We will start by defining the set of types for the simply typed λ-calculus.

Definition 2.2.1. *Let $\mathbb{V}_{\mathcal{T}}$ be an infinite set of type variables. The set of simple types, $\mathbb{T}_c$ is inductively defined from $\mathbb{V}_{\mathcal{T}}$ in the following way:*

$$\alpha \in \mathbb{V}_{\mathcal{T}} \Rightarrow \alpha \in \mathbb{T}_c$$

$$\tau, \tau' \in \mathbb{T}_c \Rightarrow (\tau \to \tau') \in \mathbb{T}_c$$

Notation Since the type constructor $\to$ is right associative, if $\tau_1, ..., \tau_n \in \mathbb{T}_c$, then

$$\tau_1 \to \tau_2 \to \ldots \to \tau_n$$

is an abbreviation for:

$$(\tau_1 \to (\tau_2 \to \ldots \to (\tau_{n-1} \to \tau_n))).$$

Definition 2.2.2. *If x is a term variable in $\mathbb{V}_{\mathcal{T}}$ and τ is a type in $\mathbb{T}_c$ then:*

- *A* statement *is of the form $M : \tau$, where the type τ is called the* predicate*, and the variable x is called the* subject *of the statement.*
- *A* declaration *is a statement where the subject is a term variable.*
- *A* basis Γ *is a set of declarations where all the subjects are distinct.*

Definition 2.2.3. *If* $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_2\}$ *is a basis, then:*

- Γ *is a partial function, with* domain*, denoted* $dom(\Gamma) = \{x_1, \ldots, x_n\}$ *and* $\Gamma(x_i) = \tau_i$*.*
- *We define* Γ_x *as* $\Gamma \setminus \{x : \tau\}$

Definition 2.2.4. *In the Curry Type System, we say that* M *has type* τ *given the basis* Γ*, and write*

$$\Gamma \vdash_{\mathcal{C}} M : \tau,$$

if $\Gamma \vdash_{\mathcal{C}} M : \tau$ *can be obtained from the following derivation rules:*

$$\Gamma_x \cup \{x : \tau\} \vdash_{\mathcal{C}} x : \tau \qquad (Axiom)$$

$$\frac{\Gamma_x \cup \{x : \tau_1\} \vdash_{\mathcal{C}} M : \tau_2}{\Gamma \vdash_{\mathcal{C}} \lambda x.M : \tau_1 \to \tau_2} \qquad (\to Intro)$$

$$\frac{\Gamma \vdash_{\mathcal{C}} M : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathcal{C}} N : \tau_1}{\Gamma \vdash_{\mathcal{C}} MN : \tau_2} \quad (\to Elim)$$

Example 2.2.1. *For the* λ*-term* $(\lambda xy.x)(\lambda x.x)$ *the following derivation is obtained in the Curry Simple Type System:*

$$\dfrac{\dfrac{\dfrac{\{x : \alpha \to \alpha, y : \beta\} \vdash_{\mathcal{C}} x : \alpha \to \alpha}{\{x : \alpha \to \alpha\} \vdash_{\mathcal{C}} \lambda y.x : \beta \to \alpha \to \alpha}}{\vdash_{\mathcal{C}} \lambda xy.x : (\alpha \to \alpha) \to \beta \to \alpha \to \alpha} \qquad \dfrac{\{x : \alpha\} \vdash_{\mathcal{C}} x : \alpha}{\vdash_{\mathcal{C}} \lambda x.x : \alpha \to \alpha}}{\vdash_{\mathcal{C}} (\lambda xy.x)(\lambda x.x) : \beta \to \alpha \to \alpha}$$

Theorem 2.2.1. ***(Subject Reduction)*** *Let* M *be a* λ*-term, and* $M \to_\beta^* M'$*, then*

$$\Gamma \vdash_{\mathcal{C}} M : \tau \Rightarrow \Gamma \vdash_{\mathcal{C}} M' : \tau$$

The implication in the other direction, called *subject expansion*, does not hold hold for this system. Take, for example, $(\lambda xy, y)(\lambda z.zz) \to_\beta^* (\lambda y.y)$, then $(\lambda y.y)$ is typable, and $(\lambda xy.y)(\lambda z.zz)$ is not.

Theorem 2.2.2. ***Strong normalization*** *Let* M *be a* λ*-term.*

$$\Gamma \vdash_{\mathcal{C}} M : \tau \Rightarrow M \textit{ is strongly normalisable.}$$

Notice that, the implication in the other direction does not hold. There are many strongly normalisable λ-terms that are not typable in this system. For example, the term $\lambda x.xx$ is not typable in the Curry Simple Type System (since to type the subterm xx, the variable x as to be both of type α and $\alpha \to \beta$) although it is strongly normalisable since it is in normal form.

$$\frac{\rho(x) = \mathcal{C}}{(\langle x, \rho\rangle, \mathcal{S}) \Rightarrow (\mathcal{C}, \mathcal{S})}\,(Var)$$

$$\frac{}{(\langle \lambda x.M, \rho\rangle, \mathcal{C} : \mathcal{S}) \Rightarrow (\langle M, \rho[x \mapsto \mathcal{C}]\rangle, \mathcal{S})}\,(Abs)$$

$$\frac{}{(\langle (M\ N), \rho\rangle, \mathcal{S}) \Rightarrow (\langle M, \rho\rangle, \langle N, \rho\rangle : \mathcal{S})}\,(App)$$

Table 2.2: Krivine Stack Machine

2.3 Operational Semantics and Abstract Machine

In this section we will present two operational semantics for λ-calculus following call-by-name and call-by-value evaluations, respectively. We will also present an abstract machine with a call-by-name evaluation.

2.3.1 Call-by-Name and Call-by-Value

Since different reduction strategies have already been introduced, we will now present evaluations for a call-by-name and call-by-value reduction strategy.
The operational semantics with call-by-name evaluation follows the rules presented bellow:

$$\frac{\mathcal{V} \text{ is a value}}{\mathcal{V} \Downarrow \mathcal{V}}\,Value \qquad \frac{S \Downarrow \lambda x.U \quad U[T/x] \Downarrow \mathcal{V}}{S\,T \Downarrow \mathcal{V}}\,App$$

The operational semantics with call-by-value evaluation follow the rules presented bellow:

$$\frac{\mathcal{V} \text{ is a value}}{\mathcal{V} \Downarrow \mathcal{V}}\,Value \qquad \frac{S \Downarrow \lambda x.U \quad T \Downarrow \mathcal{V}' \quad U[\mathcal{V}'/x] \Downarrow \mathcal{V}}{S\,T \Downarrow \mathcal{V}}\,App$$

Once again, the only difference relies on the application rule, which, in this last case, evaluates the second argument before applying the substitution. Note that the set of values $\mathcal{V}$ is composed by variables and λ-abstractions.

2.3.2 Krivine Machine

Since later on we will present two stack machines for the **PCF** language and System $\mathcal{L}_{rec}$, in this section will be presented an introduction to the Krivine stack machine

[Krivine 07], a simple lazy machine for the λ-calculus.
The Krivine Machine is a simple stack-based call-by-name evaluator restricted to λ-terms. The evaluation of this machine follows a call-by-name strategy, since on the application rule $(M\ N)$, the machine pushes the term N on the stack unevaluated, and only when the associated variable is referenced does it evaluate the later.
In the following definition for the Krivine machine, a state is defined as a pair $(\mathcal{C}, \mathcal{S})$, where $\mathcal{S}$ is a stack of λ-terms and $\mathcal{C}$ is a *closure*, that pairs free variables with environments in order to bind them, since subterms may contain free variables. The environment ρ, of those closures is a function that associates variables to closures.

Definition 2.3.1 (Krivine Configuration)**.** *The state of the Krivine machine is a pair* $(\mathcal{C}, \mathcal{S})$*, such that:*

$$\begin{array}{rl} \mathcal{C} ::= \langle M, \rho \rangle, & \textit{where } M \textit{ is a } \lambda\textit{-term} \\ \rho ::= \emptyset \mid [x \mapsto \mathcal{C}], & \textit{where } \emptyset \textit{ is the empty environment} \end{array}$$

We consider the following special cases for states:

$$\begin{array}{rll} \textit{Inicial State :} & (\langle M, \emptyset \rangle, \emptyset) & \\ \textit{Final State :} & (\langle \mathcal{V}, \emptyset \rangle, \emptyset), & \textit{where } \mathcal{V} \textit{ is a Value or a non-reducible term} \end{array}$$

The execution of the machine emulates the following rules:

- Given a **variable**: if this variable is in the environment ρ then the machine returns the closure associated with it;
- Given an **abstraction $(\lambda x.M)$ with a closure on top of the stack**: it pushes the closure in the environment associated with the variable of the abstraction (x) and then evaluates M with that environment;
- Given an **application $(M\ N)$**: it pushes the argument N associated with the environment of the application to the stack and evaluates M with that same environment.

The execution consists in constantly updating the closure and the stack. The operational semantics of the Krivine machine can be seen in Table 2.2, where for every pair of states s, t, there is a rule such that $s \Rightarrow t$.
If an execution sequence does not end in a value $\mathcal{V}$, then the final state will be the

last reduction possible. Note that the *Abs* rule corresponds to the substitution rule (Definition 2.1.4).

Example 2.3.1. *To better understand the execution of the machine, we will present an example using the term* $(\lambda x.(\lambda y.y)\ x)\ 2$*:*

$$
\begin{gathered}
(\langle((\lambda x.(\lambda y.y)\ x)\ 2), \emptyset\rangle, \emptyset) \\
\Downarrow App \\
(\langle(\lambda x.(\lambda y.y)\ x), \emptyset\rangle, \langle 2, \emptyset\rangle : \emptyset) \\
\Downarrow Abs \\
(\langle((\lambda y.y)\ x), [x \mapsto \langle 2, \emptyset\rangle]\rangle, \emptyset) \\
\Downarrow App \\
(\langle(\lambda y.y), [x \mapsto \langle 2, \emptyset\rangle]\rangle, \langle x, [x \mapsto \langle 2, \emptyset\rangle]\rangle : \emptyset) \\
\Downarrow Abs \\
(\langle y, [y \mapsto \langle x, [x \mapsto \langle 2, \emptyset\rangle]\rangle]\rangle, \emptyset) \\
\Downarrow Var \\
(\langle x, [x \mapsto \langle 2, \emptyset\rangle]\rangle, \emptyset) \\
\Downarrow Var \\
(\langle 2, \emptyset\rangle, \emptyset)
\end{gathered}
$$

The Krivine machine is considered to be inefficient due to the repeated evaluations of the same operand. However, it is a well-known standard machine for languages implementing call-by-name evaluation, which is the case of **PCF**. Based on the Krivine machine we will present two stack machines, for the **PCF** language and for System $\mathcal{L}_{rec}$.

Another well-known stack machine is the SECD [Landin 64], designed for call-by-value evaluation. We will not implement a call-by-value evaluator since we are interested in comparing the semantics of two languages following a call-by-name strategy.

Chapter 3

PCF

In this section we will describe the **PCF** language (Programming Language for Computable Functions), a variant of the typed λ-calculus with the addition of ground types **int** and **bool** and a restricted conversion relation. The implementation of this language, using Haskell [Has 03], will also be described.
The **PCF** language was originally introduced by Gordon Plotkin [Plotkin 77], inspired by the language LCF (Language for Computable Functions [Scott 93]).

3.1 Syntax

We will start by defining the set of types for the **PCF** language.

Definition 3.1.1. *The set of **PCF** types, $\mathbb{T}_p$, is inductively defined in the following way:*

$$\begin{array}{rcl} \alpha \in \{\textbf{\textit{int}}, \textbf{\textit{bool}}\} & \Rightarrow & \alpha \in \mathbb{T}_p \\ \tau, \tau' \in \mathbb{T}_p & \Rightarrow & (\tau \to \tau') \in \mathbb{T}_p \end{array}$$

Identically to the notation described in Chapter 2, we will assume that types associate to the right.
For each type $\tau \in \mathbb{T}_p$, we have an infinite set of typed variables and a collection of typed constants:

$$\mathbb{V}_\tau = \{x^\tau, y^\tau, z^\tau, \ldots\} \qquad \mathbb{C}_\tau = \{\mathbf{c}^\tau\}$$

We will now define the set of constants for the **PCF** language.

Definition 3.1.2. *The set of constants, $\mathbb{C}_\tau$, is defined as follows:*

- $\bar{n}$*:* ***int*** *for* $n = 0, 1, 2, \ldots$
- tt, ff*:* ***bool***
- succ, pred*:* $\boldsymbol{int} \to \boldsymbol{int}$
- iszer*:* $\boldsymbol{int} \to \boldsymbol{bool}$

And for each type $\tau \in \mathbb{T}_p$:

- cond_τ*:* $\boldsymbol{bool} \to \tau \to \tau \to \tau$
- Y_τ*:* $(\tau \to \tau) \to \tau$

Definition 3.1.3. *The set Λ_p of typed* ***PCF*** *terms $M : \tau$, with $\tau \in \mathbb{T}_p$ is defined as:*

$$\begin{array}{rcll} x^\tau : \tau \in \Lambda_p & & & \textit{(variables)} \\ \boldsymbol{c}^\tau : \tau \in \Lambda_p & & & \textit{(constants)} \\ M : \tau_1 \in \Lambda_p & \Rightarrow & \lambda x^\tau.M : \tau \to \tau_1 \in \Lambda_p & \textit{(abstraction)} \\ M : \tau_1 \to \tau_2 \in \Lambda_p,\ N : \tau_1 \in \Lambda_p & \Rightarrow & MN : \tau_2 \in \Lambda_p & \textit{(application)} \end{array}$$

Again, we assume that application associates to the left.
Some examples of **PCF** terms:

- $\mathsf{pred}\,\bar{6}$: $\mathbf{int}$
- $\mathsf{cond}_{\mathbf{int}}\ (\mathsf{iszer}\ (\mathsf{succ}\,\bar{2}))\,\bar{6}$: $\mathbf{int} \to \mathbf{int}$

As another example, consider the following recursive definition of *factorial*:

```
f(x)=if x=0 then 1 else x*f(x-1)
```

One can easily encode it as a **PCF** term:

$$\mathsf{Y}_{\mathbf{int}}(\lambda f^{\mathbf{int}\to\mathbf{int}}.\lambda x^{\mathbf{int}}.\mathsf{cond}_{\mathbf{int}}\ (\mathsf{iszer}\ x)\ \bar{1}\ (\mathsf{mult}\ x\ (f\ (\mathsf{pred}\ x))))$$

Where,

$$\mathsf{mult} = \mathsf{Y}_{\mathbf{int}}(\lambda f^{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}.\lambda m^{\mathbf{int}}.\lambda n^{\mathbf{int}}.\mathsf{cond}_{\mathbf{int}}\ (\mathsf{iszer}\ m)\ 0\ (\mathsf{add}\ (f\ (\mathsf{pred}\ m)\ n)\ n)$$

$$(\lambda x^{\tau}.M)N \Rightarrow M[N/x] \qquad \mathsf{Y}_{\tau}M \Rightarrow M(Y_{\tau}M)$$

$$\frac{M \Rightarrow M'}{MN \Rightarrow M'N}$$

$$\frac{M \Rightarrow M'}{\mathsf{succ}\ M \Rightarrow \mathsf{succ}\ M'} \qquad \mathsf{succ}\ \overline{n} \Rightarrow \overline{n+1}$$

$$\frac{M \Rightarrow M'}{\mathsf{pred}\ M \Rightarrow \mathsf{pred}\ M'} \qquad \mathsf{pred}\ \overline{0} \Rightarrow \overline{0} \qquad \mathsf{pred}\ \overline{n+1} \Rightarrow \overline{n}$$

$$\frac{M \Rightarrow M'}{\mathsf{cond}\ MN_1N_2 \Rightarrow \mathsf{cond}\ M'N_1N_2} \qquad \mathsf{cond}\ \mathsf{tt}\ MN \Rightarrow M \qquad \mathsf{cond}\ \mathsf{ff}\ MN \Rightarrow N$$

Table 3.1: PCF Call-by-Name Evaluation

$$\mathsf{add} = \mathsf{Y}_{\mathbf{int}}(\lambda f^{\mathbf{int}\to\mathbf{int}\to\mathbf{int}}.(\lambda m^{\mathbf{int}}.(\lambda n^{\mathbf{int}}.\mathsf{cond}_{\mathbf{int}}(\mathsf{iszer}\ m)\ n\ (\mathsf{succ}\ (f\ (\mathsf{pred}\ m)\ n)))))$$

We can also have higher-order programs, e.g.:

$$\mathsf{iter}\ f\ n\ x = \begin{cases} x & \text{if } n = 0 \\ f(\mathsf{iter}(n-1)fx) & \text{if } n > 0 \end{cases}$$

Which we can encode in **PCF** as:

$$\mathsf{Y}_{\mathbf{int}}(\lambda F^{\sigma}.\lambda n^{\mathbf{int}}.\lambda f^{\tau\to\tau}.\lambda x^{\tau}.\mathsf{cond}_{\tau}\ (\mathsf{iszer}\ n)\ x^{\tau}\ (f(F(\mathsf{pred}\ n)\ f\ x)))$$

where $\sigma \stackrel{def}{=} \mathbf{int} \to (\tau \to \tau) \to \tau \to \tau$ and $\tau, \sigma \in \mathbb{C}$.
The correctness of such encodings can be easily proved by induction.
The evaluation of terms in **PCF** is defined by the transition relation in Table 3.1. Note that, in **PCF**, we do not have the following evaluation rules:

$$\frac{M \Rightarrow M'}{\lambda x^{\tau}.M \Rightarrow \lambda x^{\tau}.M'} \qquad \frac{N \Rightarrow N'}{MN \Rightarrow MN'}$$

3.2 Implementation

In Figure 3.1, one can see a diagram of the **PCF** implementation, starting by an analysis of the language, followed by a call-by-name and abstract machine interpreters, which evaluate the expression into a normal form, if one exists.
The implementation of this language started with a parser in Happy [Gill 01], that transformed the **PCF** input into a data structure in Haskell (Code 3.1). Then,

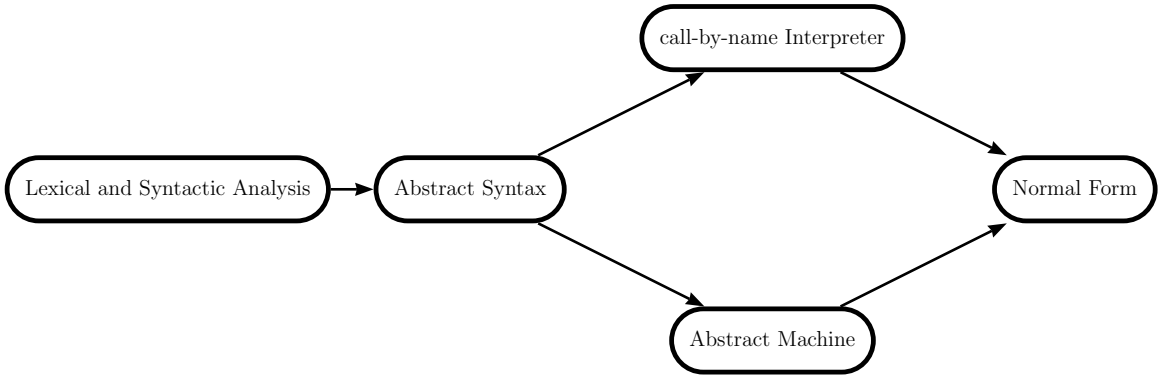

Figure 3.1: **PCF** Implementation

a call-by-name evaluation function was created, and finally the stack machine was implemented.

Code 3.1: PCF Language Data Structure

```
ExpPCF :: = Var String | Nat Int | TT | FF
  | IsZ | Succ | Pred | Cond | PtoF
  | Lambd String ExpPCF
  | App ExpPCF ExpPCF
      deriving (Show,Eq)
```

In the data structure created in Haskell (Code 3.1), the structures Var String and Nat Int, hold the names of the variables and the values of the integers, respectively; the TT and FF serve as the values tt and ff of the **PCF** language; the PtoF is the equivalent of **PCF** fixed point, Y. For the application and λ-terms, the structures Lambd String ExpPCF and App ExpPCF ExpPCF were created.
Finally, the structures IsZ, Succ, Pred and Cond will serve as the **PCF** constants iszer, succ, pred and cond.

3.2.1 Call-by-Name Interpreter

The call-by-name evaluation function, which complete code can be seen in Appendix (Code A.1), will be explained in several parts.
This function receives a **PCF** expression, returning the evaluation of that same expression. We started by implementing the evaluation of values (Code 3.2) that cannot be reduced, which means that the function will return them as they are.

Code 3.2: PCF call-by-name evaluation for values

```
eval :: ExpPCF -> ExpPCF
eval (Var a)      = (Var a)
eval (Lambd x m) = (Lambd x m)
eval (Nat x)      = (Nat x)
eval Succ         = Succ
eval Pred         = Pred
eval Cond         = Cond
eval IsZ          = IsZ
eval PtoF         = PtoF
```

We consider as values variables, abstractions and the **PCF** constants iszer, pred, succ, cond, Y. And since these **PCF** functions were implemented as values and do not receive arguments, we use the application to evaluate them with a term. For example, the term cond $B\,M\,N$ is translated to App (App (App Cond b)m)n, as one can see in Code 3.3.

The eval function evaluates following the rules on Table 3.1.

Code 3.3: PCF call-by-name evaluation for application

```
eval (App PtoF m) = eval(App m (App PtoF m))
eval (App (Lambd x m) n) = eval (substt m x n)
eval (App (App (App Cond TT) m) n) = eval m
eval (App (App (App Cond FF) m) n) = eval n
eval (App (App (App Cond b) m) n) = eval(App(App(App Cond(eval b)) m) n)
eval (App Succ m) = case (eval m) of
    (Nat n)   -> (Nat (n+1))
    t         -> (App Succ t)
eval (App Pred m) = case (eval m) of
    (Nat (n+1)) -> (Nat n)
    t           -> (App Pred t)
eval (App IsZ (Nat n)) = if (n==0) then TT else FF
eval (App IsZ m) = eval(App IsZ (eval m))
eval (App s t)   = eval_aux(App (eval s) t)
eval t           = t
```

In order to implement the call-by-name evaluation function, there was the necessity to create a substitution function (Definition 2.1.4, Code A.2) for the application of a λ-term.

(app)	$(M\ N,\ \mathcal{S}_p)$	$\Rightarrow$	$(M, N{:}\mathcal{S}_p)$
(abs)	$((\lambda x.M), N{:}\mathcal{S}_p)$	$\Rightarrow$	$(M[N/x],\ \mathcal{S}_p)$
(fixed point)	$(\mathsf{Y}, M{:}\mathcal{S}_p)$	$\Rightarrow$	$(M,\ (\mathsf{Y}\ M){:}\mathcal{S}_p)$
(cond1)	$(\mathsf{cond},\ M{:}N_1 : N_2{:}\mathcal{S}_p)$	$\Rightarrow$	$(M, \mathsf{COND}(N_1, N_2){:}\mathcal{S}_p)$
(cond2)	$(\mathsf{tt},\ (\mathsf{COND}\ (N_1, N_2)){:}\mathcal{S}_p)$	$\Rightarrow$	$(N_1, \mathcal{S}_p)$
(cond3)	$(\mathsf{ff},\ (\mathsf{COND}\ (N_1, N_2)){:}\mathcal{S}_p)$	$\Rightarrow$	$(N_2, \mathcal{S}_p)$
(succ1)	$(\overline{n},\ \mathsf{SUCC}{:}\mathcal{S}_p)$	$\Rightarrow$	$(\overline{n+1}, \mathcal{S}_p)$
(succ2)	$(\mathsf{succ}, M{:}\mathcal{S}_p)$	$\Rightarrow$	$(M, \mathsf{SUCC}{:}\mathcal{S}_p)$
(pred1)	$(\overline{0}, \mathsf{PRED}{:}\mathcal{S}_p)$	$\Rightarrow$	$(\overline{0}, \mathcal{S}_p)$
(pred2)	$(\overline{n+1}, \mathsf{PRED}{:}\mathcal{S}_p)$	$\Rightarrow$	$(\overline{n}, \mathcal{S}_p)$
(pred3)	$(\mathsf{pred}, M{:}\mathcal{S}_p)$	$\Rightarrow$	$(M, \mathsf{PRED}{:}\mathcal{S}_p)$
(iszer1)	$(\overline{0}, \mathsf{ISZER}{:}\mathcal{S}_p)$	$\Rightarrow$	$(\mathsf{tt}, \mathcal{S}_p)$
(iszer2)	$(\overline{n+1}, \mathsf{ISZER}{:}\mathcal{S}_p)$	$\Rightarrow$	$(\mathsf{ff}, \mathcal{S}_p)$
(iszer3)	$(\mathsf{iszer}, M{:}\mathcal{S}_p)$	$\Rightarrow$	$(M, \mathsf{ISZER}{:}\mathcal{S}_p)$

Table 3.2: PCF Stack Machine

Code 3.4: Auxiliary Evaluation Function

```
1 eval_aux :: ExpPCF -> ExpPCF
  eval_aux (App (Lambd x u) t) = eval(substt u x t)
3 eval_aux (App s           t) = (App s t)
```

We also created an auxiliary function (Code 3.4) that looks at the first term of the application: if it is a λ-term, it evaluates the substitution for that term, if it is not, and since it has already been evaluated and is therefore in normal form, our auxiliary function returns the application term as it is, because the call-by-name evaluation of the application does not reduce the second term. This way, we avoid the problem of trying to evaluate the first term when it is not evaluable.

3.2.2 Stack Machine

In this last section we will show how the **PCF** language can be implemented as a stack machine.
We will start by defining the elements of the stack followed by the definition of the machine configuration.

Definition 3.2.1 (Stack Machine Elements)**.** *Let $\mathcal{S}_p$ be the set of the elements of the stack machine, which is an extension of the **PCF** terms, then:*

$$\begin{array}{rcl} \alpha \in \Lambda_{\mathcal{P}} & \Rightarrow & \alpha \in \mathcal{S}_p \\ M, N \in \Lambda_{\mathcal{P}} & \Rightarrow & \mathsf{COND}(M, N) \in \mathcal{S}_p \end{array}$$

In order to make the call-by-name evaluation possible in the machine, the following terms are also added:

$$\mathsf{SUCC},\ \mathsf{PRED},\ \mathsf{ISZERO} \in \mathcal{S}_p$$

Definition 3.2.2 (**PCF** Stack Machine)**.** *The state of the PCF stack machine is a pair* $(M, \mathcal{S}_p)$*, such that:*

- M *is a* ***PCF*** *term*
- $\mathcal{S}_p$ *is a stack of* ***PCF*** *extended terms*

We consider the following special cases for states:

$$\begin{array}{rl} \textit{Inicial State :} & (M, \epsilon) \\ \textit{Final State :} & (\mathcal{V}, \mathcal{S}_p), \quad \textit{where } \mathcal{V} \textit{ is a Value} \end{array}$$

The basic principle of the machine is to find the next redex, using the stack $\mathcal{S}_p$ to store future computations. And for every pair of states s, t, there is a rule, based on a call-by-name strategy, such that $s \Rightarrow t$ (Table 3.2).
The execution of the stack terminates when a final state is reached, which means that no other reduction is possible.

Example 3.2.1. *To better understand the execution of the machine, we will present an example using the term* $((\lambda m.(\lambda n.(((\mathsf{cond}\ (\mathsf{iszer}\ m))\ n)(\mathsf{pred}\ m))))\ 2)\ 3$*:*

$$\begin{array}{c} (((\lambda m.(\lambda n.(((\mathsf{cond}\ (\mathsf{iszer}\ m))\ n)(\mathsf{pred}\ m))))\ 2)\ 3, \epsilon) \\ \Downarrow \textit{app} \\ ((\lambda m.(\lambda n.(((\mathsf{cond}\ (\mathsf{iszer}\ m))\ n)(\mathsf{pred}\ m))))\ 2), 3 : \epsilon) \\ \Downarrow \textit{app} \\ (\lambda m.(\lambda n.(((\mathsf{cond}\ (\mathsf{iszer}\ m))\ n)(\mathsf{pred}\ m))), 2 : 3 : \epsilon) \\ \Downarrow \textit{abs} \\ ((\lambda n.(((\mathsf{cond}\ (\mathsf{iszer}\ 2))\ n)(\mathsf{pred}\ 2))), 3 : \epsilon) \\ \Downarrow \textit{abs} \\ (((\mathsf{cond}\ (\mathsf{iszer}\ 2))\ 3)(\mathsf{pred}\ 2), \epsilon) \\ \Downarrow \textit{app} \\ (((\mathsf{cond}\ (\mathsf{iszer}\ 2))\ 3), (\mathsf{pred}\ 2) : \epsilon) \end{array}$$

$$\Downarrow app$$
$$((\mathsf{cond}\ (\mathsf{iszer}\ 2)), 3 : (\mathsf{pred}\ 2) : \epsilon)$$
$$\Downarrow app$$
$$(\mathsf{cond}\ , (\mathsf{iszer}\ 2) : 3 : (\mathsf{pred}\ 2) : \epsilon)$$
$$\Downarrow cond1$$
$$((\mathsf{iszer}\ 2), \mathsf{COND}(3, (\mathsf{pred}\ 2)) : \epsilon)$$
$$\Downarrow app$$
$$(\mathsf{iszer}\ , 2 : \mathsf{COND}(3, (\mathsf{pred}\ 2)) : \epsilon)$$
$$\Downarrow iszer3$$
$$(2, \mathsf{ISZER} : \mathsf{COND}(3, (\mathsf{pred}\ 2)) : \epsilon)$$
$$\Downarrow iszer2$$
$$(\mathsf{ff}, \mathsf{COND}(3, (\mathsf{pred}\ 2)) : \epsilon)$$
$$\Downarrow cond3$$
$$((\mathsf{pred}\ 2), \epsilon)$$
$$\Downarrow app$$
$$(\mathsf{pred}\ , 2 : \epsilon)$$
$$\Downarrow pred3$$
$$(2, \mathsf{PRED} : \epsilon)$$
$$\Downarrow pred2$$
$$(1, \epsilon)$$

Considering that later on, we will show a stack machine for System $\mathcal{L}_{rec}$ (Section 4.3), there was also the necessity to implement it on **PCF** in order to make a better comparison between the two languages.

Code 3.5: Data Structure for the Stack

```
type Stck = [ExpStck]
data ExpStck = E ExpPCF
       | COND (ExpPCF,ExpPCF)
       deriving (Show)
```

In order to implement the stack machine function and to represent the new terms associated with it, a data structure was created (Code 3.5), in which the ExpStck is the structure that will hold the elements of the stack machine $\mathcal{S}_p$.
One can see the complete implementation in the Appendix Code A.3.
The implementation of this stack machine will prove useful once we start to compare results between the **PCF** language and System $\mathcal{L}_{rec}$, which we will describe in the next chapter.

Chapter 4

System $\mathcal{L}_{rec}$

We will now introduce System $\mathcal{L}_{rec}$, a syntactically linear λ-calculus extended with numbers, pairs and an unbounded recursor that preserves the syntactic linearity of the calculus, introduced in [Alves 11].

4.1 Syntax

As an extension of λ-calculus, the $\mathcal{L}_{rec}$ system also uses the notion of substitution to model the definition and application of functions. The representations of the application and the abstraction are also the same.

Definition 4.1.1 ($\mathcal{L}_{rec}$)**.** *Let $\mathbb{V}$ be an infinite set of variables. The set of $\mathcal{L}_{rec}$ terms $\Lambda_{\mathcal{R}}$, is built in the following way:*

(Zero)	0	$\Rightarrow$	$0 \in \Lambda_{\mathcal{R}}$	
(Variables)	$x \in \mathbb{V}$	$\Rightarrow$	$x \in \Lambda_{\mathcal{R}}$	
(Application)	$M, N \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(MN) \in \Lambda_{\mathcal{R}}$,	*if* $FV(M) \cap FV(N) = \emptyset$
(Abstraction)	$x \in \mathbb{V},\ M \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(\lambda x.M) \in \Lambda_{\mathcal{R}}$,	*if* $x \in FV(M)$
(Successor)	$M \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(\mathsf{S}\ M) \in \Lambda_{\mathcal{R}}$	
(Pairs)	$M, N \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(\langle M, N \rangle) \in \Lambda_{\mathcal{R}}$,	*if* $FV(M) \cap FV(N) = \emptyset$
(Let)	$x, y \in \mathbb{V},\ M, N \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(\mathsf{let}\ \langle x, y \rangle = M\ in\ N) \in \Lambda_{\mathcal{R}}$,	*if* $x, y \in FV(N)$, $x \neq y$, $FV(M) \cap FV(N) = \emptyset$
(Rec)	$M_1, M_2, M_3, M_4 \in \Lambda_{\mathcal{R}}$	$\Rightarrow$	$(\mathsf{rec}\ M_1\ M_2\ M_3\ M_4) \in \Lambda_{\mathcal{R}}$,	*if* $FV(M_i) \cap FV(M_j) = \emptyset$, *for* $i \neq j$

Once again, we consider the application left associative and the abstraction right associative.

4.1.1 Variables and Substitution

We will now formalize the concepts of *free* and *bound* variables of a term for System $\mathcal{L}_{rec}$.

Definition 4.1.2. ***(Free Variables)*** *Let $M \in \Lambda_{\mathcal{R}}$, the set $FV(M)$ of free variables of M is inductively defined as follows:*

$$\begin{array}{rcl} FV(0) & = & \emptyset \\ FV(x) & = & \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) \\ FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\ FV(\mathsf{S}\, M) & = & FV(M) \\ FV(\langle M, N\rangle) & = & FV(M) \cup FV(N) \\ FV(\mathsf{let}\ \langle x, y\rangle = M\ in\ N) & = & FV(M) \cup (FV(U) \setminus \{x, y\}) \\ FV(\mathsf{rec}\, T\, U\, V\, W) & = & FV(T) \cup FV(U) \cup FV(V) \cup FV(W) \end{array}$$

Definition 4.1.3. ***(Bound Variables)*** *Let $M \in \Lambda_{\mathcal{R}}$, the set $BV(M)$ of bound variables of M is inductively defined as follows:*

$$\begin{array}{rcl} BV(0) & = & \emptyset \\ BV(x) & = & \emptyset \\ BV(MN) & = & BV(M) \cup BV(N) \\ BV(\lambda x.M) & = & BV(M) \cup \{x\} \\ BV(\mathsf{S}\, M) & = & \emptyset \\ BV(\langle M, N\rangle) & = & BV(M) \cup BV(N) \\ BV(\mathsf{let}\ \langle x, y\rangle = M\ in\ N) & = & BV(M) \cup (BV(U) \cup \{x, y\}) \\ BV(\mathsf{rec}\, T\, U\, V\, W) & = & BV(T) \cup BV(U) \cup BV(V) \cup BV(W) \end{array}$$

Note that regarding the names of free and bound variables, this system, similar to the λ-calculus in Chapter 2, assumes the Barendregt's convention.

Definition 4.1.4. ***($\mathcal{L}_{rec}$ Substitution)*** *The result of substituting the free occurrences of x by L in M (denoted by $M[L/x]$) is defined as:*

$$0[L/y] \equiv 0$$

$$x[L/y] \equiv \begin{cases} L & \textit{if } x \equiv y \\ x & \textit{otherwise} \end{cases}$$

$$(M\ N)[L/y] \equiv (M[L/y])\ (N[L/y])$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} (\lambda x.M) & \textit{if } x \equiv y \\ \lambda x.(M[L/y]) & \textit{otherwise} \end{cases}$$

$$(\langle M, N\rangle)[L/y] \equiv \langle M[L/y], N[L/y]\rangle$$

$$(\mathsf{S}\ M)[L/y] \equiv \mathsf{S}\ (M[L/y])$$

$$(\mathsf{let}\ \langle x, y\rangle = M\ in\ N)[L/z] \equiv \begin{cases} (\mathsf{let}\ \langle x, y\rangle = M\ in\ N) & \textit{if } x \equiv z \vee y \equiv z \\ (\mathsf{let}\ \langle x, y\rangle = M[L/y]\ in\ N[L/y]) & \textit{otherwise} \end{cases}$$

$$(\mathsf{rec}\ T\ U\ V\ W)[L/y] \equiv (\mathsf{rec}\ (T[L/y])\ \ (U[L/y])\ \ (V[L/y])\ \ (W[L/y]))$$

Since it is a linear language, the substitution could be done taking into consideration the rules for free variables which could restrict the substitution to one branch of the term.
The reduction rules for this system consist of a restriction of the β-reduction from λ-calculus (Section 2.1.3) for closed terms, a rule for *Let* and two rules for the recursor as shown in the following definition.

Definition 4.1.5 ($\mathcal{L}_{rec}$ Reduction). *The following contraction rules define the notion of reduction on System $\mathcal{L}_{rec}$:*

$$\begin{array}{lrll}
\beta: & (\lambda x.M)N & \to M[N/x], & \textit{if } FV(N) = \emptyset \\
\textit{Let}: & (\mathsf{let}\ \langle x, y\rangle = \langle M, N\rangle\ in\ T) & \to (T[M/x])[N/y], & \textit{if } FV(M) = FV(N) = \emptyset \\
\textit{Rec}: & (\mathsf{rec}\ \langle 0, T\rangle\ U\ V\ W) & \to U, & \textit{if } FV(T) = FV(V) = FV(W) = \emptyset \\
\textit{Rec}: & (\mathsf{rec}\ \langle \mathsf{S}\ T_1, T_2\rangle\ U\ V\ W) & \to V(\mathsf{rec}\ (W\ \langle T_1, T_2\rangle)\ U\ V\ W), & \textit{if } FV(V) = FV(W) = \emptyset
\end{array}$$

The linearity is preserved by reduction since the *Rec* rules are only triggered when the conditions for free variables hold. These rules pattern-match on a pair of numbers whereas the usual bounded recursor works on a single number, because we are representing both bounded and unbounded recursion with the same operator. The pair

$\langle M, N\rangle$ for this recursor represents the value being tested $M = f(N)$ and the value N that produced M. In this way, we can preserve the last iteration number more efficiently without interfering with the computation of the function.
The last parameter in the recursor is used to compute the next pair of numbers, so we can program unbounded and bounded recursion.
We will now give some examples, that later on will prove useful in the compilation from **PCF** to $\mathcal{L}_{rec}$.

Example 4.1.1 (**Projections and duplication of natural numbers**). *Defining projections of pairs $\langle a, b\rangle$ of natural numbers, is easy using recursion:*

$$pr_1 = \lambda x.\mathsf{let}\ \langle a, b\rangle\ =\ x\ \mathsf{in}\ (\mathsf{rec}\ \langle b, 0\rangle\ a\ (\lambda x.x)\ (\lambda x.x))$$

$$pr_2 = \lambda x.\mathsf{let}\ \langle a, b\rangle\ =\ x\ \mathsf{in}\ (\mathsf{rec}\ \langle a, 0\rangle\ b\ (\lambda x.x)\ (\lambda x.x))$$

And the function for copying numbers:

$$C = \lambda x.\mathsf{rec}\ \langle x, 0\rangle\ \langle 0, 0\rangle\ (\lambda x.\mathsf{let}\ x\ =\ \langle a, b\rangle\ \mathsf{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)\ (\lambda x.x)$$

We also define some arithmetic functions, which will be used later on in the compilation.

Example 4.1.2 (**Arithmetic Functions**). *The arithmetic functions can be encoded in $\mathcal{L}_{rec}$ in the following way:*

- ***add*** $= \lambda mn.\mathsf{rec}\ \langle m, 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)\ (\lambda x.x)$;
- ***mult*** $= \lambda mn.\mathsf{rec}\ \langle m, 0\rangle\ 0\ (\mathsf{add}\ n)$;
- ***pred*** $= \lambda n.pr_1\ (\mathsf{rec}\ \langle n, 0\rangle\ \langle 0, 0\rangle\ \boldsymbol{F}\ (\lambda x.x))$
  where $\boldsymbol{F} = \lambda x.\mathsf{let}\ \langle t, 0\rangle\ = C(pr_2\ x)\ \mathsf{in}\ \langle t, \mathsf{S}\ u\rangle$;
- ***iszero*** $= \lambda n.pr_1\ (\mathsf{rec}\ \langle n, 0\rangle\ \langle 0, \mathsf{S}\ 0\rangle\ (\lambda x.C(pr_2\ x))\ (\lambda x.x))$

These functions will be very useful later on in this thesis in order to compare results between **PCF** and $\mathcal{L}_{rec}$.
The correctness of these encodings can be easily proved by induction.

4.1.2 Types for System $\mathcal{L}_{rec}$

We will start by defining the *linear types* of System $\mathcal{L}_{rec}$.

Definition 4.1.6. *Let $\mathbb{T}_{\mathcal{R}}$ be the set of $\mathcal{L}_{rec}$ types defined inductively in the following way:*

$$\begin{array}{rcl} & & \boldsymbol{nat} \in \mathbb{T}_{\mathcal{R}} \\ \tau_0, \tau_1 \in \mathbb{T}_{\mathcal{R}} & \Rightarrow & (\tau_0 \multimap \tau_1) \in \mathbb{T}_{\mathcal{R}} \\ \tau_0, \tau_1 \in \mathbb{T}_{\mathcal{R}} & \Rightarrow & (\tau_0 \otimes \tau_1) \in \mathbb{T}_{\mathcal{R}} \end{array}$$

*Where **nat** is a type for natural numbers.*

Definition 4.1.7. *Let Γ be a type environment for System $\mathcal{L}_{rec}$. We say that M has type τ given a basis Γ, and write:*

$$\Gamma \vdash_{\mathcal{L}} M \,:\, \tau,$$

if $\Gamma \vdash_{\mathcal{L}} M \,:\, \tau$ can be obtained from the derivation rules shown in Table 4.1.

To denote the set of variables that occur in Γ, we write $dom(\Gamma)$, the same as for the λ-calculus.

Theorem 4.1.1 (Properties of reductions in $\mathcal{L}_{rec}$)**.** *The properties of reduction in $\mathcal{L}_{rec}$ are:*

1. *If $\Gamma \vdash_{\mathcal{L}} T : \tau$ then $dom(\Gamma) = FV(\tau)$.*
2. *Subject Reduction: Reductions preserves types.*
3. *Church-Rosser: System $\mathcal{L}_{rec}$ is confluent.*
4. *Adequacy: If $\vdash_{\mathcal{L}} T : \tau$ in System $\mathcal{L}_{rec}$ and T is a normal form, then there are $\mathcal{L}_{rec}$ terms U, M such that:*

$$\begin{array}{lcl} \tau = \boldsymbol{nat} & \Rightarrow & T = \mathsf{S}(\mathsf{S} \ldots (\mathsf{S}\, 0)) \\ \tau = \tau_0 \otimes \tau_1 & \Rightarrow & T = \langle U, M \rangle \\ \tau = \tau_0 \multimap \tau_1 & \Rightarrow & T = \lambda x.M \end{array}$$

5. *System $\mathcal{L}_{rec}$ is not strongly normalizing, even for typable terms.*

Axiom:

$$\frac{}{\{x\ :\ \tau_0\} \vdash_{\mathcal{L}} x\ :\ \tau_0}\ (Axiom)$$

Logical Rules:

$$\frac{\Gamma_x \cup \{x : \tau_0\} \vdash_{\mathcal{L}} T : \tau_1}{\Gamma \vdash_{\mathcal{L}} \lambda x.T : \tau_0 \multimap \tau_1}\ (\multimap Intro) \qquad \frac{\Gamma \vdash_{\mathcal{L}} T : \tau_0 \multimap \tau_1 \qquad \triangle \vdash_{\mathcal{L}} U : \tau_0}{\Gamma \cup \triangle \vdash_{\mathcal{L}} T\,U : \tau_1}\ (\multimap Elim)$$

$$\frac{\Gamma \vdash_{\mathcal{L}} T : \tau_0 \quad \triangle \vdash_{\mathcal{L}} U : \tau_1}{\Gamma \cup \triangle \vdash_{\mathcal{L}} \langle T, U\rangle : \tau_0 \otimes \tau_1}\ (\otimes Intro) \qquad \frac{\Gamma \vdash_{\mathcal{L}} T : \tau_0 \otimes \tau_1 \qquad \triangle_{x,y} \cup \{x : \tau_0,\ y : \tau_1\} \vdash_{\mathcal{L}} U : \tau_2}{\Gamma \cup \triangle \vdash_{\mathcal{L}} \mathsf{let}\ \langle x, y\rangle = T\ \mathsf{in}\ U : \tau_2}\ (\otimes Elim)$$

Numbers:

$$\frac{}{\vdash_{\mathcal{L}} 0 : \mathbf{nat}}\ (Zero) \qquad \frac{\Gamma \vdash_{\mathcal{L}} N : \mathbf{nat}}{\Gamma \vdash_{\mathcal{L}} \mathsf{S}\ N : \mathbf{nat}}\ (Succ)$$

$$\frac{\Gamma \vdash_{\mathcal{L}} T : \mathbf{nat} \otimes \mathbf{nat} \qquad \Theta \vdash_{\mathcal{L}} U : \tau_0 \qquad \triangle \vdash_{\mathcal{L}} V : \tau_0 \multimap \tau_0 \qquad \Sigma \vdash_{\mathcal{L}} W : \mathbf{nat} \otimes \mathbf{nat} \multimap \mathbf{nat} \otimes \mathbf{nat}}{\Gamma \cup \Theta \cup \triangle \cup \Sigma \vdash_{\mathcal{L}} \mathsf{rec}\ T\ U\ V\ W : \tau_0}\ (Rec)$$

Table 4.1: Type System for System $\mathcal{L}_{rec}$

The proofs for this and other properties on $\mathcal{L}_{rec}$ can be found in [Alves 11].
In the linear λ-calculus terms are consumed by reduction, even though we are not able to discard arguments of functions. In order to erase, System $\mathcal{L}_{rec}$ applies the technique of Solvability [Barendregt 84], generalizing the encoding of projections given in Example 4.1.1.
To erase a term T of type τ, $\mathcal{L}_{rec}$ uses a function Erase ($\mathcal{E}(T, \tau)$) and a function Make ($\mathcal{M}(\tau)$) to build a term of a specific type ($\mathcal{E}$ and $\mathcal{M}$ are mutually recursive).

Definition 4.1.8 (Erasing)**.** *If $\Gamma \vdash_{\mathcal{L}} T : \tau$, then $\mathcal{E}(T, \tau)$ is defined as follows:*

$$\begin{array}{lcl} \mathcal{E}(T, \boldsymbol{nat}) & = & \mathsf{rec}\ \langle T, 0\rangle\ (\lambda x.x)\ (\lambda x.x)\ (\lambda x.x) \\ \mathcal{E}(T, \tau_0 \otimes \tau_1) & = & \mathsf{let}\ \langle x, y\rangle = T\ \mathsf{in}\ (\mathcal{E}(x, \tau_0)\mathcal{E}(y, \tau_1)) \\ \mathcal{E}(T, \tau_0 \multimap \tau_1) & = & \mathcal{E}(T\ \mathcal{M}(\tau_0), \tau_1) \end{array}$$

and

$$\begin{array}{lcl} \mathcal{M}(\boldsymbol{nat}) & = & 0 \\ \mathcal{M}(\tau_0 \otimes \tau_1) & = & \langle \mathcal{M}(\tau_0), \mathcal{M}(\tau_1)\rangle \\ \mathcal{M}(\tau_0 \multimap \tau_1) & = & \lambda x.\mathcal{E}(x, \tau_0)\mathcal{M}(\tau_1) \end{array}$$

Since we are in a non-normalizing calculus, not every term can be erased in this way. In order to copy closed terms, the System $\mathcal{L}_{rec}$ as also defined a duplication function:

$$\frac{\mathcal{V} \text{ is a value}}{\mathcal{V} \Downarrow \mathcal{V}}\, Val \qquad \frac{S \Downarrow \lambda x.U \quad U[T/x] \Downarrow \mathcal{V}}{S\,T \Downarrow \mathcal{V}}\, App \qquad \frac{T \Downarrow \langle T_1, T_2\rangle \quad (\lambda xy.U)T_1T_2 \Downarrow \mathcal{V}}{\mathsf{let}\ \langle x, y\rangle = T\ \mathsf{in}\ U \Downarrow \mathcal{V}}\, Let$$

$$\frac{T \Downarrow \langle T_1, T_2\rangle \quad T_1 \Downarrow 0 \quad U \Downarrow \mathcal{V}}{\mathsf{rec}\,T\,U\,V\,W \Downarrow \mathcal{V}}\, Rec_1 \qquad \frac{T \Downarrow \langle T_1, T_2\rangle \quad T_1 \Downarrow S\,T' \quad v(\mathsf{rec}\,(W\langle T', T_2\rangle)\,U\,V\,W) \Downarrow \mathcal{V}}{\mathsf{rec}\,T\,U\,V\,W \Downarrow \mathcal{V}}\, Rec_2$$

Table 4.2: CBN evaluation for System $\mathcal{L}_{rec}$

Definition 4.1.9 (Duplication)**.** *$D^\tau : \tau \multimap \tau \otimes \tau$ is defined as:*

$$\lambda x.\mathsf{rec}\,\langle \mathsf{S}(\mathsf{S}\,0), 0\rangle\,\langle \mathcal{M}(\tau), \mathcal{M}(\tau)\rangle\,\boldsymbol{F}\,(\lambda x.x)$$

where $\boldsymbol{F} = (\lambda y.\mathsf{let}\,\langle z, w\rangle\ =\ y\ \mathsf{in}\ \mathcal{E}(z, \tau)\langle w, x\rangle)$.

These definitions will prove useful in order to define the compilation from **PCF** to System $\mathcal{L}_{rec}$.
Results concerning the properties of these functions were proved in [Alves 11].

4.1.3 Evaluation Strategies

We will define two evaluation strategies for the $\mathcal{L}_{rec}$ system, call-by-name and call-by-value, and derive a simple stack machine.

Call-by-Name The call-by-name evaluation function for the $\mathcal{L}_{rec}$ system is presented in Table 4.2. When a closed term T evaluates to a value, in System $\mathcal{L}_{rec}$ we write $T \Downarrow \mathcal{V}$. In this system, a value, or a *weak normal form*, consists of the following terms: 0, $\mathsf{S}T$, $\lambda x.T$ and $\langle S, T\rangle$. In System $\mathcal{L}_{rec}$ the symbol S is used as a constructor of natural numbers, and therefore is not evaluated, unlike **PCF** that used the successor as a function. Also note that no closedness condition is needed in the evaluation rules, since we are only considering closed terms.
Since in System $\mathcal{L}_{rec}$ we write the *Let* rule using application, the two strategies only differ in the application rule.

Call-by-Value Similar to the λ-calculus, the call-by-value evaluation changes only in the application rule:

$$\frac{S \Downarrow \lambda x.U \quad T \Downarrow V' \quad U[V'/x] \Downarrow \mathcal{V}}{S\,T \Downarrow \mathcal{V}}$$

Since the *Rec* and *Let* rules rely on the application rule, there is no need to change them for the call-by-value strategy.
Unlike CBN, the CBV strategy does not always reach a value, even if a close term has one.

4.2 Implementation

Similar to the **PCF** language, the diagram of System $\mathcal{L}_{rec}$ implementation in Figure 4.1, starts by an analysis of the language, followed by interpreters call-by-name, call-by-value and an abstract machine, that follows a call-by-name evaluation. The final goal of this implementation is for the expression to reach its normal form, if one exists.

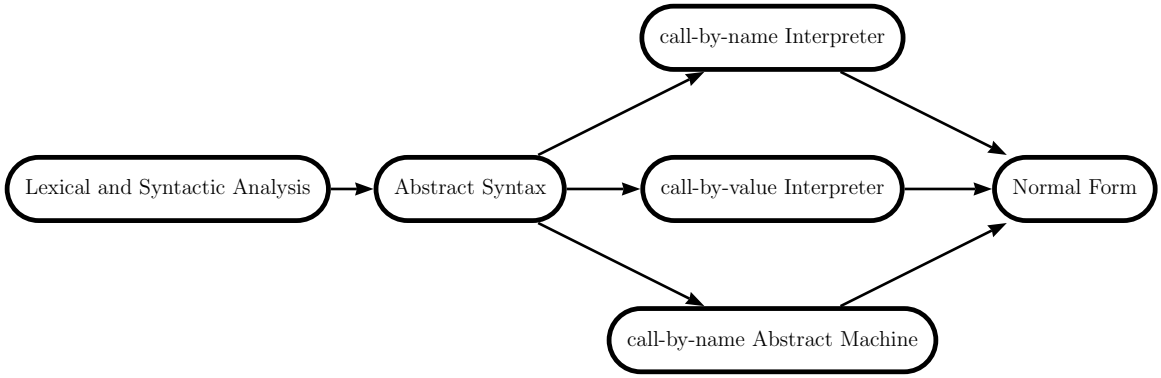

Figure 4.1: System $\mathcal{L}_{rec}$ Implementation

Following the same implementation of the **PCF** language, the System $\mathcal{L}_{rec}$ implementation started with a parser, also written in Happy which converts the input, $\mathcal{L}_{rec}$ terms, into a data structure, Code 4.1.

Code 4.1: Data Structure for the $\mathcal{L}_{rec}$ System

```
data Exp =
        Rec Exp Exp Exp Exp
      | Pair Exp Exp
      | Let (String,String) Exp Exp
      | Var String
      | Zero
      | Suc Exp
      | Lambd String Exp
      | App Exp Exp
      deriving (Show, Eq)
```

This structure was built according to the needs of the language:

- (Zero) represents the only numerical number in $\mathcal{L}_{rec}$, 0;
- (Suc Exp), (Var String), (Lambd String Exp), (Pair Exp Exp) represent the values of the system: successor, variables, abstraction and pairs, respectively;
- (Rec Exp Exp Exp Exp) represents the rec term;
- the (Let (String,String) Exp Exp) structure was created with a tuple of strings instead of a data structure pair, in order to make the implementation of the system in Haskell easier;
- (App Exp Exp) obviously represents the term application MN.

In the following subsections we will present the implementations of the call-by-name and call-by-value evaluations and a stack machine for the $\mathcal{L}_{rec}$ system, using this data structure.

4.2.1 Call-by-Name Interpreter

The call-by-name implementation, was done according to Table 4.2. We will discuss here some aspects of the implementation. One can see the complete code for that function in the Appendix (Code A.4).

Code 4.2: The call-by-name Evaluation for System $\mathcal{L}_{rec}$

```
eval_cbn (Suc a)
  | (value a)        = (Suc a)
  | otherwise        = (Suc (eval_cbn a))
```

In Code 4.2 one can see that the evaluation of the successor term was done using a function called value :: Exp -> Bool which is used to determine if the expression within the successor is a value or not. Although in the call-by-name evaluation rules of System $\mathcal{L}_{rec}$ the evaluation within the successor was not described, when implementing we felt the need to evaluate under successor since we realized that the evaluation of the terms might not end in a normal form if we did not evaluate inside the successor. As for the other values, when eval_cbn receives them, it returns them as they are, since those expressions are already in their normal form.

Code 4.3: Application Evaluation for System $\mathcal{L}_{rec}$

```
eval_cbn (App (Lambd x u) t) = eval_cbn(substt u x t)
eval_cbn (App s t) = cbn_aux(App (eval_cbn s) t)
```

There are two rules for the application (Code 4.3):

- *Application of a λ-abstractor*: where the substitution function is used (Definition A.7, Appendix Code A.7) to evaluate following the rules in Table 4.2;
- *Application of any other two terms*: where we use the auxiliary function (Code 4.4) in order to evaluate the application without causing the same problem described in Section 3.2.1.

Code 4.4: Auxiliary Evaluation Function for CBN

```
cbn_aux :: Exp -> Exp
cbn_aux (App (Lambd x u) t) = eval_cbn(substt u x t)
cbn_aux (App s      t) = (App s t)
```

In the let expression, if we do not have a pair (Code 4.5: line 3), we first evaluate that term in order to reach the pair, and then make the correct let evaluation (Code 4.5: line 1) according to Table 4.2.

Code 4.5: call-by-name Evaluation Function for Let expression

```
1 eval_cbn (Let (x, y) (Pair t1 t2) u) =
              eval_cbn (App (App (Lambd x (Lambd y (u))) t1) t2)
3 eval_cbn (Let (a, b) p  u) = eval_cbn (Let (a, b) (eval_cbn p) u)
```

Finally, the rec expression has several different cases:

- *The pair starts with the value*:
  - Zero (Code 4.6: line 1): then we return the term U (Table 4.2: Rec_1);
  - Suc t (Code 4.6: line 2): the successor rule applies, so we use the recursion rule (Table 4.2: Rec_2).
- *The pair is not reduced to a value*: so we call the function again in order to evaluate the first term of the pair (Table 4.2: Rec_1);
- *There is no pair*: we call the function recursively in order to find the pair so we can apply the evaluation (Table 4.2).

Code 4.6: call-by-name Evaluation Function for Ret expression

```
eval_cbn (Rec (Pair (Zero) t2) u v w) = eval_cbn u
eval_cbn (Rec (Pair (Suc t) t2) u v w) =
              eval_cbn (App v (Rec (App w (Pair t t2)) u v w))
eval_cbn (Rec (Pair p_1 p_2) u v w) =
        eval_cbn (Rec (Pair (eval_cbn p_1) p_2) u v w)
eval_cbn (Rec p u v w) = eval_cbn (Rec (eval_cbn p) u v w)
```

If none of the above rules applies to the expression, then the expression is already in normal form, and the evaluation terminates.

4.2.2 Call-by-Value Interpreter

As it was mentioned in Section 5.2, the call-by-value function changes only in the application rule. Therefore, the function is identical to the one described above, except for the application rule (Code 4.7) where we evaluate the second term before calling the substitution function.

Code 4.7: call-by-value Evaluation Function

```
eval_cbv (App (Lambd x u) t) = eval_cbv(substt u x (eval_cbv t))
eval_cbv (App s           t) = cbv_aux(App (eval_cbv s) t)
```

In this function we also used an auxiliary function (Code A.6: cbv_aux :: Exp -> Exp), once again, to prevent continually trying to evaluate something that is already in its normal form.

4.2.3 Stack Machine

We will start by defining the elements of the stack machine.

Definition 4.2.1 (System $\mathcal{L}_{rec}$ Stack Machine Elements)**.** *Let $\mathcal{S}_{\mathcal{R}}$ be the set of elements of the stack machine, which is an extension of the System $\mathcal{L}_{rec}$ terms, $\Lambda_{\mathcal{R}}$, then:*

(app)	$(MN,\, \mathcal{S})$	$\Rightarrow$	$(M,\, N : \mathcal{S})$	
(abs)	$((\lambda x.M),\, N : \mathcal{S})$	$\Rightarrow$	$(M[N/x],\, \mathcal{S})$	
(let)	$(\mathsf{let}\langle x,\, y\rangle = N \,\mathsf{in}\, M,\, \mathcal{S})$	$\Rightarrow$	$(N,\, \mathsf{LET}(x, y, M) : \mathcal{S})$	
(pair1)	$(\langle N_1,\, N_2\rangle,\, \mathsf{LET}(x, y, M) : \mathcal{S})$	$\Rightarrow$	$(M[N_1/x][N_2/y],\, \mathcal{S})$	
(rec)	$(\mathsf{rec}\, N\, U\, V\, W,\, \mathcal{S})$	$\Rightarrow$	$(N, \mathsf{REC}(U, V, W) : \mathcal{S})$	
(pair2)	$(\langle N_1,\, N_2\rangle,\, \mathsf{REC}(U, V, W) : \mathcal{S})$	$\Rightarrow$	$(N_1, \mathsf{REC}'(N_2, U, V, W) : \mathcal{S})$	
(zero)	$(0,\, \mathsf{REC}'(T, U, V, W) : \mathcal{S})$	$\Rightarrow$	$(U,\, \mathcal{S})$	
(rec1)	$((\mathsf{S}\, N),\, \mathsf{REC}'(T, U, V, W) : \mathcal{S})$	$\Rightarrow$	$(V,\, (\mathsf{rec}\, (W\, \langle N, T\rangle)\, U\, V\, W) : \mathcal{S})$	
(succ1)	$((\mathsf{S}\, N),\, \mathsf{SUCC} : \mathcal{S})$	$\Rightarrow$	$(N,\, \mathsf{SUCC} : \mathsf{SUCC} : \mathcal{S})$	$N \not\equiv \mathcal{V}$
(succ2)	$((\mathsf{S}\, N),\, \mathcal{S})$	$\Rightarrow$	$(N,\, \mathsf{SUCC} : \mathcal{S})$	$N \not\equiv \mathcal{V}$
(succ3)	$(N,\, \mathsf{SUCC} : \mathcal{S})$	$\Rightarrow$	$(\mathsf{S}\, N,\, \mathcal{S})$	$N \equiv \mathcal{V}$

Table 4.3: $\mathcal{L}_{rec}$ Stack Machine

$$
\begin{array}{rcl}
 & & \mathsf{SUCC} \in \mathcal{S}_{\mathcal{R}} \\
M \in \Lambda_{\mathcal{R}} & \Rightarrow & M \in \mathcal{S}_{\mathcal{R}} \\
x, y \in \mathbb{V} \text{ and } T \in \Lambda_{\mathcal{R}} & \Rightarrow & LET(x, y, T) \in \mathcal{S}_{\mathcal{R}} \\
U, V, W \in \Lambda_{\mathcal{R}} & \Rightarrow & REC(U, V, W) \in \mathcal{S}_{\mathcal{R}} \\
N, U, V, W \in \Lambda_{\mathcal{R}} & \Rightarrow & REC'(N, U, V, W) \in \mathcal{S}_{\mathcal{R}}
\end{array}
$$

As it can be seen in Table 4.3, the stack machine for System $\mathcal{L}_{rec}$ is based on the call-by-name strategy.

Definition 4.2.2 ($\mathcal{L}_{rec}$ Stack Configuration)**.** *The $\mathcal{L}_{rec}$ Stack Machine is composed by a states of the form $(M, \mathcal{S})$, such that:*

- $M \in \Lambda_{\mathcal{R}}$
- $\mathcal{S}$ *is a stack of extended terms,* $\mathcal{S}_{\mathcal{R}}$

We consider the following special cases for states:

$$
\begin{array}{rl}
\textit{Inicial State :} & (M, \epsilon) \\
\textit{Final State :} & (\mathcal{V}, \epsilon), \quad \textit{where } \mathcal{V} \textit{ is a } \mathcal{L}_{rec} \textit{ Value}
\end{array}
$$

The basic principle of the machine is the same as for **PCF**: find the next redex, using the stack $\mathcal{S}$ to store future computations.
Although the Krivine machine (Table 2.3.2) included an environment, in this case none is needed since, when the $\mathcal{L}_{rec}$ Stack Machine finds a binding of a variable to a term, replaces the unique occurrence of that variable, i.e., it replaces the occurrence

of the variable by the term.
The $\mathcal{L}_{rec}$ Stack Machine ends when a normal form is reached, or, in case it can't be reached, with the last reduction possible.

Example 4.2.1. *To better understand the execution of the machine, we will present an example using the term* $((\lambda m.(\lambda n.\mathsf{rec}\ \langle m, 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)(\lambda x.x)))\ (\mathsf{S}\ 0))\ (\mathsf{S}\ (\mathsf{S}\ 0))$*:*

$$((((\lambda m.(\lambda n.\mathsf{rec}\ \langle m, 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)(\lambda x.x)))\ (\mathsf{S}\ 0))\ (\mathsf{S}\ (\mathsf{S}\ 0)), \epsilon)$$
$$\Downarrow \textit{app}$$
$$(((\lambda m.(\lambda n.\mathsf{rec}\ \langle m, 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)(\lambda x.x)))\ (\mathsf{S}\ 0)), (\mathsf{S}\ (\mathsf{S}\ 0)) : \epsilon)$$
$$\Downarrow \textit{app}$$
$$((\lambda m.(\lambda n.\mathsf{rec}\ \langle m, 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)(\lambda x.x))), (\mathsf{S}\ 0) : (\mathsf{S}\ (\mathsf{S}\ 0)) : \epsilon)$$
$$\Downarrow \textit{abs}$$
$$((\lambda n.\mathsf{rec}\ \langle(\mathsf{S}\ 0), 0\rangle\ n\ (\lambda x.\mathsf{S}\ x)(\lambda x.x)), (\mathsf{S}\ (\mathsf{S}\ 0)) : \epsilon)$$
$$\Downarrow \textit{abs}$$
$$(\mathsf{rec}\ \langle(\mathsf{S}\ 0), 0\rangle\ (\mathsf{S}\ (\mathsf{S}\ 0))\ (\lambda x.\mathsf{S}\ x)(\lambda x.x), \epsilon)$$
$$\Downarrow \textit{rec}$$
$$(\langle(\mathsf{S}\ 0), 0\rangle, \mathsf{REC}((\mathsf{S}\ (\mathsf{S}\ 0)), (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \epsilon)$$
$$\Downarrow \textit{pair2}$$
$$((\mathsf{S}\ 0), \mathsf{REC}'(0, (\mathsf{S}\ (\mathsf{S}\ 0)), (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \epsilon)$$
$$\Downarrow \textit{succ}$$
$$((\lambda x.\mathsf{S}\ x), (\mathsf{rec}\ ((\lambda x.x)\langle 0, 0\rangle)(\mathsf{S}\ (\mathsf{S}\ 0))\ (\lambda x.\mathsf{S}\ x)\ (\lambda x.x)) : \epsilon)$$
$$\Downarrow \textit{abs}$$
$$((\mathsf{S}(\mathsf{rec}\ ((\lambda x.x)\langle 0, 0\rangle)(\mathsf{S}\ (\mathsf{S}\ 0))\ (\lambda x.\mathsf{S}\ x)\ (\lambda x.x))), \epsilon)$$
$$\Downarrow \textit{succ1}$$
$$((\mathsf{rec}\ ((\lambda x.x)\langle 0, 0\rangle)(\mathsf{S}\ (\mathsf{S}\ 0))\ (\lambda x.\mathsf{S}\ x)\ (\lambda x.x)), \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{rec}$$
$$(((\lambda x.x)\langle 0, 0\rangle), \mathsf{REC}(\mathsf{S}(\mathsf{S}\ 0), (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{app}$$
$$((\lambda x.x), \langle 0, 0\rangle : \mathsf{REC}(\mathsf{S}(\mathsf{S}\ 0), (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{abs}$$
$$(\langle 0, 0\rangle, \mathsf{REC}(\mathsf{S}(\mathsf{S}\ 0), (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{pair2}$$
$$(0, \mathsf{REC}'(0, (\lambda x.\mathsf{S}\ x), (\lambda x.x)) : \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{zero}$$
$$(\mathsf{S}(\mathsf{S}\ 0), \mathsf{SUCC} : \epsilon)$$
$$\Downarrow \textit{succ2}$$
$$((\mathsf{S}\ 0), \mathsf{SUCC} : \mathsf{SUCC} : \epsilon)$$

$$\Downarrow succ2$$
$$(0, \mathsf{SUCC} : \mathsf{SUCC} : \mathsf{SUCC} : \epsilon)$$
$$\Downarrow succ3$$
$$(\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ 0)), \epsilon)$$

The implementation of the stack machine was done according to Table 4.3, using the data structure shown below in Code 4.8, where we created the new terms that were described (Description 4.2.1): $\mathsf{LET}(x, y, t)$ (line 3); $\mathsf{REC}(u, v, w)$ (line 4) and $\mathsf{REC}'(n, u, v, w)$ (line 5); and also a term (SUCC) that will help in the stack evaluation of the successor, as in the call-by-name and call-by-value evaluations, where we had to evaluate inside the term.
The structure (E Exp) was created in order to include the System $\mathcal{L}_{rec}$ terms in the new extended set of the stack, ExpStck.

Code 4.8: Data Structure for the Stack

```
type Stck = [ExpStck]
data ExpStck = E Exp
       | LET (String,String,Exp)
       | REC (Exp, Exp, Exp)
       | RECC (Exp, Exp, Exp, Exp)
       | SUCC
       deriving (Show)
```

The complete code for this implementation can be found in the Appendix (Code A.8), but some of it will be better explained next.

Code 4.9: Successor Evaluation in the Stack Machine

```
stck ((Suc n), SUCC:s)                    =
     if (value n) then stck (Suc (Suc n), s) else stck (n, SUCC:SUCC:s)
stck ((Suc n), s)                         =
     if (value n) then ((Suc n),s) else stck (n, SUCC:s)
stck (n, SUCC:s)                          =
     if (value n) then stck ((Suc n),s) else ((Suc n),s)
```

We will start by explaining the need for the term SUCC. As one can see in Code 4.9, first we verify if the term inside the successor is a value or not. If it is not a value, then we add the new data structure SUCC to the stack, so we can first evaluate inside the successor, and then rebuild the successor term. If it is a value, then there is nothing to evaluate and the machine just needs to rebuild the successor term.
The rest of the stack implementation is straightforward, if one follows Table 4.3. Note that we use the substitution function (Definition 2.1.4, Code A.7), in the application

of a λ-term.

In the next chapter, we will compare the results from this stack machine with the **PCF** stack machine.

Chapter 5

Compiling

In this chapter we will show how the **PCF** language can be encoded in System $\mathcal{L}_{rec}$. We will start by defining the compilation from **PCF** to $\mathcal{L}_{rec}$, following with the implementation of the compilation and finally, we will compare evaluation results between both. We will also present a variant of System $\mathcal{L}_{rec}$ with built-in natural numbers, its implementation and compare its evaluation results with the previous ones.

5.1 Compiling

We will start by defining how the **PCF** types and environments can be translated into System $\mathcal{L}_{rec}$ types.

Definition 5.1.1. ***PCF*** *types and environments are translated into System* $\mathcal{L}_{rec}$ *types using* $\langle . \rangle$*:*

$$\begin{aligned} \langle \textbf{\textit{int}} \rangle &= \textbf{\textit{nat}} \\ \langle \textbf{\textit{bool}} \rangle &= \textbf{\textit{nat}} \\ \langle \tau_1 \to \tau_2 \rangle &= \langle \tau_1 \rangle \multimap \langle \tau_2 \rangle \\ \langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle &= x_1 : \langle \tau_1 \rangle, \ldots, x_n : \langle \tau_n \rangle \end{aligned}$$

Using the encoding for types and environments, we will now show how a **PCF** program can be encoded into System $\mathcal{L}_{rec}$. In the following abbreviations, the variables x_1 and x_2 are assumed fresh, and $[x]T$ will be defined bellow.

$$C^{x_1,x_2}_{x:\tau} T \;=\; \mathsf{let}\ \langle x_1, x_2 \rangle = D^{\tau} x \ \mathsf{in}\ T$$

$$
\begin{array}{lcl}
\langle tt\rangle & = & 0 \\
\langle ff\rangle & = & \mathsf{S}\,0 \\
\langle n\rangle & = & \mathsf{S}^n 0 \\
\langle \mathsf{succ}\rangle & = & \lambda n.\mathsf{rec}\,\langle n,0\rangle\,(\mathsf{S}\,0)\,(\lambda x.\mathsf{S}\,x)\,(\lambda x.x) \\
\langle \mathsf{pred}\rangle & = & \lambda n.pr_1(\mathsf{rec}\,\langle n,0\rangle\,\langle 0,0\rangle\,(\lambda x.\mathsf{let}\,\langle t,u\rangle = D^{\mathbf{nat}}(pr_2\,x)\,\mathsf{in}\,\langle t,\mathsf{S}\,u\rangle)\,(\lambda x.x)) \\
\langle \mathsf{iszero}\rangle & = & \lambda n.pr_1(\mathsf{rec}\,\langle n,0\rangle\,\langle 0,\mathsf{S}\,0\rangle\,(\lambda x.D^{\mathbf{nat}}(pr_2\,x))\,(\lambda x.x)) \\
\langle \mathsf{Y}_\tau\rangle & = & \lambda f.\mathsf{rec}\,\langle \mathsf{S}\,0,0\rangle\,\mathcal{M}(\langle\tau\rangle)\,f\,(\lambda x.\mathsf{let}\,\langle y,z\rangle = x\,\mathsf{in}\,\langle \mathsf{S}\,y,z\rangle) \\
\langle \mathsf{cond}_\tau\rangle & = & \lambda tuv.\mathsf{rec}\,\langle t,0\rangle\,u\,(\lambda x.(\mathsf{rec}\,\langle 0,0\rangle\,(\lambda x.x)\,x\,(\lambda x.x))\,v)\,(\lambda x.x) \\
\langle x\rangle & = & x \\
\langle UV\rangle & = & \langle U\rangle\langle V\rangle \\
\langle \lambda x^\tau.T\rangle & = & \begin{cases} \lambda x.[x^\tau]\langle T\rangle & \text{if } x\in \mathrm{FV}(T) \\ \lambda x.(\mathsf{rec}\,\langle 0,0\rangle\,(\lambda x.x)\,x\,(\lambda x.x))\langle T\rangle & \text{otherwise} \end{cases}
\end{array}
$$

Table 5.1: PCF compilation into $\mathcal{L}_{rec}$

$$A^x_y T = ([x]T)[y/x]$$

Definition 5.1.2 (Linearization)**.** *Let T be a **PCF** term, with $FV(T) = \{x_1,\ldots,x_n\}$ and $x_1 : \tau_1,\ldots,x_n : \tau_n \vdash T : \tau$. The compilation into System $\mathcal{L}_{rec}$ is defined as: $[x_1^{\tau_1}]\ldots[x_n^{\tau_n}]\langle T\rangle$, where $\langle\bullet\rangle$ is defined in Table 5.1, and for a term T and a variable x, such that $x \in FV(T)$, $[x]T$ is inductively defined in the following way:*

$$
\begin{array}{lcl}
[x](\mathsf{S}\,U) & = & \mathsf{S}([x]U) \\
[x]x & = & x \\
[x](\lambda y.U) & = & \lambda y.[x]U \\
[x^\tau](M\,U) & = & \begin{cases} C^{x_1,x_2}_{x:\tau}(A^{x_1}_x\,M)(A^{x_2}_x\,U) & x\in FV(M)\cap FV(U) \\ ([x]M)U & x\notin FV(U) \\ M([x]U) & x\notin FV(M) \end{cases}
\end{array}
$$

In this definition, $[x]T$, is not defined for the entire syntax of System $\mathcal{L}_{rec}$, because although others syntactic constructors may appear in T, these are the result of $\langle\bullet\rangle$ and are therefore closed terms were x does not occur free. The complete encoding can be seen in Table 5.1.

Also note that $\langle\mathsf{succ}\rangle$ is not encoded as $\lambda x.\mathsf{S}\,x$, since System $\mathcal{L}_{rec}$ originally does not evaluate under abstractions or S. Although when the implementation was completed, there was the necessity to evaluate inside the successor, the encoding was preserved since $\mathsf{cond}_\tau(\mathsf{succ}(\mathsf{Y}_\rho\,(\lambda x.x)))\;P\;Q$ is $\langle\mathsf{cond}_\tau\rangle(\langle\mathsf{succ}\rangle(\langle\mathsf{Y}_\rho\rangle\;(\lambda x.x)))\;\langle P\rangle\;\langle Q\rangle$, so if we encoded $\langle succ\rangle$ into $\lambda x.\mathsf{S}\,x$ we would have $\langle Q\rangle$ which is not correct.

In the second case of the abstraction, we use a recursor over zero which returns

the identity function discarding the argument. The variable x is used directly as a parameter of the function, since when we implemented the encoding we noted that, we only used integers types, **int** and **bool**, the use of the erasing function was unjustified. We will now give an example of a compilation from **PCF** to System $\mathcal{L}_{rec}$.

Example 5.1.1. *Using Table 5.1 the **PCF** term,* $(\mathsf{cond}_{\boldsymbol{int}}\,(\mathsf{iszer}\,m)\,\overline{1}\,(\mathsf{pred}\,m))$*, can be encoded into System* $\mathcal{L}_{rec}$ *as:*

$$((\lambda tuv.\mathsf{rec}\,\langle t,u\rangle\,u\,(\lambda x.(\mathsf{rec}\,\langle 0,0\rangle\,(\lambda x.x)\,x(\lambda x.x))\,v)\,(\lambda x.x))$$
$$(\lambda n.\,pr_1\,(\mathsf{rec}\,\langle n,0\rangle\,\langle 0,\mathsf{S}\,0\rangle(\lambda x.\,D^{\boldsymbol{nat}}(pr_2\,x))\,(\lambda x.x))m)\;(\mathsf{S}\,0)$$
$$(\lambda n.\,pr_1\,(\mathsf{rec}\,\langle n,0\rangle\,\langle 0,0\rangle\,(\lambda x.\mathsf{let}\,\langle t,u\rangle = \,D^{\boldsymbol{nat}}(pr_2\,x)\,\mathsf{in}\,\langle t,\mathsf{S}\,u\rangle)\,(\lambda x.x)))\,m)$$

The implementation starts by changing the data structure of **PCF** terms so that its names in Haskell would not be the same as the $\mathcal{L}_{rec}$ terms. In that sense, the new data structure for **PCF** expressions can be seen in Code 5.1.

Code 5.1: **PCF** Terms

```
data ExpPCF = VarPCF String | NatPCF Int | TT | FF
      | IsZ | Succ | Pred | Cond
      | PtoF Type
      | LambdPCF (String,Type) ExpPCF
      | AppPCF ExpPCF ExpPCF
      deriving (Show,Eq)
```

The complete code for the encoding function can be seen in Appendix Code A.9, where the function linearFV (Code A.10) was done according to Definition 5.1.2, where Type is a new Haskell type created to encode **PCF** types into $\mathcal{L}_{rec}$ types.
Note that the functions erase and make were also implemented in Appendix Code A.11 and A.12; and the copy function is the one used in Definition 5.1.2 for copying the variables.

5.2 Comparing Results

In order to compare the results between the two languages we decided to implement an abstract time measure from [Lago 06]: with the purpose of giving an cost evaluator for the λ-calculus where elementary reductions steps are counted proportionally to the number of corresponding steps in a Turing machine.
We will start by defining how the cost for the abstract time measure will be calculated, where we denote the size of a λ-term M as $|M|$.

Definition 5.2.1 (Abstract Time Measure)**.** *An abstract time measure can be defined as:*

$$M \twoheadrightarrow^{\epsilon} N \qquad \frac{M \to N \qquad n = max\{1,\ |N| - |M|\}}{M \twoheadrightarrow^{n} N} \qquad \frac{M \twoheadrightarrow^{\alpha} N \qquad N \twoheadrightarrow^{\beta} L}{M \twoheadrightarrow^{\alpha \cdot \beta} L}$$

Were $\alpha \cdot \beta$ denotes the concatenation of $\alpha, \beta \in \mathbb{N}^*$, and given $\alpha = n_1 \cdot \ldots \cdot n_m \in \mathbb{N}^*$, we define $||\alpha|| = \Sigma_{i=1}^{m} n_i$.

To implement this definition, we started by creating a function to calculate the size of a $\mathcal{L}_{rec}$ expression (Code 5.2), and similarly for the **PCF** language in Appendix Code A.13.

Code 5.2: $\mathcal{L}_{rec}$ expression size

```
esizeLRec :: Exp -> Int
esizeLRec (Zero)          = 1
esizeLRec (Var a)         = 1
esizeLRec (Suc a)         = 1+(esizeLRec a)
esizeLRec (Pair n m)      = (esizeLRec m)+(esizeLRec n)
esizeLRec (Lambd x t)     = 1+(esizeLRec t)
esizeLRec (App m n)       = (esizeLRec m)+(esizeLRec n)
esizeLRec (Let (x,y) p u) = 2+(esizeLRec p)+(esizeLRec u)
esizeLRec (Rec t u v w)   =
        (esizeLRec t)+(esizeLRec u)+(esizeLRec v)+(esizeLRec w)
```

After the function for calculating the expression size, we changed the call-by-name evaluation function, so that instead of returning just an evaluated expression, it returns a tuple with the evaluated expression, the number of reductions that took to evaluate it and the cost defined in Definition 5.2.1. The complete code for the new evaluation function can be seen in the Appendix (Code A.14 for System $\mathcal{L}_{rec}$ and Code A.15 for the **PCF** language).
The initial arguments of the evaluation function are the expression to be evaluated and the counter, starting as zero. Every time a reduction is made, the function adds one number to the reduction counter, calculates the cost and adds it to the previous cost count, in order to calculate $\alpha \cdot \beta$ (Definition 5.2.1).

Code 5.3: $\mathcal{L}_{rec}$ evaluation with Abstract Time Measure

```
eval_cbn (App s          t) n =
        let (e1,nr1,it) = (eval_cbn s n)
            (e2,nr2,it2) = (cbn_aux(App e1 t) nr1)
        in (e2, nr2,it2+it)
eval_cbn (Let (a, b)  p          u) n =
```

```
6           let (e2,nr2,it) = (eval_cbn p n)
                (e1,nr1,it2) = (eval_cbn (Let (a, b) e2 u) nr2)
8   eval_cbn (Rec (Pair p_1 p_2)       u v w) n =
        let (e2,nr2,it) = (eval_cbn p_1 n)
10          (e1,nr1,it2) = (eval_cbn (Rec (Pair e2 p_2) u v w) nr2)
        in (e1,nr1,it2+it)
12  eval_cbn (Rec p                    u v w) n =
        let (e2,nr2,it) = (eval_cbn p n)
14          (e1,nr1,it2) = (eval_cbn (Rec e2 u v w) nr2)
        in (e1,nr1,it2+it)
```

As one can see in Code 5.3, in some cases of the System $\mathcal{L}_{rec}$ evaluation function, the number of reductions is not updated and the cost is not calculated, since a reduction is not actually made. Although we still make the final sum of the costs and number of reductions from previous reductions.
Note that when the expression is a value, the function also does not add on the number of reductions or calculate the cost.

Code 5.4: **PCF** language evaluation with Abstract Time Measure

```
1   eval (App Succ m) n = case (eval m n) of
                               ((Nat n),nr,c)    -> ((Nat (n+1)),(nr+1),c+1)
3                              (t,nr,c)          -> ((App Succ t),nr,c)
    eval (App Pred m) n = case (eval m n) of
5                      ((Nat (n+1)),nr,c) -> ((Nat n),(nr+1),c+1)
                       (t,nr,c)           -> ((App Pred t),nr,c)
7   eval (App IsZ (Nat n)) nr = if (n==0)
                                then (TT,(nr+1),1)
9                               else (FF,(nr+1),1)
```

In the evaluation function for the **PCF** language, although the counter for reductions is increased, some cases do not calculate the cost even when a reduction is made (Code 5.4), that happens due to the fact that the difference between the expressions $|N| - |M|$ would always be negative, so the maximum between one and the difference would always be one, therefore, we avoid having to calculate the cost by replacing it with the number one.
Also note that this function has special cases, as it was for System $\mathcal{L}_{rec}$ where no new counter or cost is calculated (Code 5.5).

Code 5.5: **PCF** language evaluation with Abstract Time Measure

```
1   eval (App (App (App Cond b) m) n)  nr =
      let (e1,nr1,c1) = (eval b nr)
```

	Abstract Time Measure			Number of reductions		
Function	**PCF**	$\mathcal{L}_{rec}$	**PCF to** $\mathcal{L}_{rec}$	**PCF**	$\mathcal{L}_{rec}$	**PCF to** $\mathcal{L}_{rec}$
add 2 3	56	34	4975	23	11	503
mult 2 3	247	108	5091	87	37	3012
fib 4	1255	402	289499	197	116	18356
fact 4	22165	3530	5464324	4546	915	345722

Table 5.2: Results Comparison

```
3         (e2,nr2,c2) = (eval (App (App (App Cond e1) m) n) nr1)
    in (e2,nr2,c2+c1)
5 eval (App IsZ m) n = let (e1,nr1,c) = (eval m n)
                           (e2,nr2,c2) = (eval(App IsZ e1) nr1)
7                      in (e2,nr2,c+c2)
  eval (App s t) n = let (e1,nr1,c) = (eval s n)
9                        (e2,nr2,c2) = (eval_aux(App e1 t) nr1)
                     in (e2,nr2,c+c2)
```

After this implementation some tests were made to the program in **PCF**, System $\mathcal{L}_{rec}$ and to the one with the encoding from **PCF** to $\mathcal{L}_{rec}$. The results can be seen in Table 5.2 were the functions called add, mult, fib and fact, are encodings of the addition, multiplication, fibonacci and factorial functions to **PCF** and System $\mathcal{L}_{rec}$, respectively. Although the results between **PCF** and System $\mathcal{L}_{rec}$ seem to be what was expected, the results with the encoding from **PCF** to $\mathcal{L}_{rec}$ could be better. Partially this could relay on the fact that $\mathcal{L}_{rec}$ does not operate on natural numbers but on successors over zero, therefore we decided to implement System $\mathcal{L}_{rec}$ with built-in natural numbers. We will call it $\mathcal{L}_{rec}^{\mathbf{nat}}$.

5.2.1 System $\mathcal{L}_{rec}$ with built-in naturals

Adding naturals to System $\mathcal{L}_{rec}$ was done by adding in the parser a rule for the numbers, but in order to really try to simplify $\mathcal{L}_{rec}$ and make it more efficient we decided to implement the predecessor and the function iszero, as it exists in **PCF**.
In that sense, the syntax of the $\mathcal{L}_{rec}^{\mathbf{nat}}$ will have the variables, application, abstraction, pairs, rec and let from the System $\mathcal{L}_{rec}$, with the addition of naturals, such that, for the set of $\mathcal{L}_{rec}^{\mathbf{nat}}$ terms, $\Lambda_{\mathcal{R}^N}$:

$$\overline{n} \in \mathbb{N} \Rightarrow \overline{n} \in \Lambda_{\mathcal{R}^N}$$

$$\frac{M \Rightarrow M'}{\mathsf{suc}\, M \Rightarrow \mathsf{suc}\, M'} \qquad \mathsf{suc}\, \overline{n} \Rightarrow \overline{n+1}$$

$$\frac{M \Rightarrow M'}{\mathsf{pre}\, M \Rightarrow \mathsf{pre}\, M'} \qquad \mathsf{pre}\, \overline{0} \Rightarrow \overline{0} \qquad \mathsf{pre}\, \overline{n+1} \Rightarrow \overline{n}$$

Table 5.3: $\mathcal{L}_{rec}^{\mathbf{nat}}$ Call-by-Name Evaluation

(app)	$(MN, \mathcal{S})$	$\Rightarrow$	$(M, N{:}\mathcal{S})$
(abs)	$((\lambda\ \mathrm{x}.M), N{:}\mathcal{S})$	$\Rightarrow$	$(M[N/x], \mathcal{S})$
(succ1)	$(\overline{n}, \mathrm{SUCC}{:}\mathcal{S})$	$\Rightarrow$	$(\overline{n+1}, \mathcal{S})$
(succ2)	$(\mathsf{suc}\ , M{:}\mathcal{S})$	$\Rightarrow$	$(M, \mathrm{SUCC}{:}\mathcal{S})$
(pred1)	$(\overline{0}, \mathrm{PRE}{:}\mathcal{S})$	$\Rightarrow$	$(\overline{0}, \mathcal{S})$
(pred2)	$(\overline{n+1}, \mathrm{PRE}{:}\mathcal{S})$	$\Rightarrow$	$(\overline{n}, \mathcal{S})$
(pred3)	$(\mathsf{pre}, M{:}\mathcal{S})$	$\Rightarrow$	$(M, \mathrm{PRE}{:}\mathcal{S})$
(iszero1)	$(\overline{0}, \mathrm{ISZERO}{:}\mathcal{S})$	$\Rightarrow$	$(\overline{0}, \mathcal{S})$
(iszero2)	$(\overline{n+1}, \mathrm{ISZERO}{:}\mathcal{S})$	$\Rightarrow$	$(\overline{1}, \mathcal{S})$
(iszero3)	$(\mathsf{iszero}, M{:}\mathcal{S})$	$\Rightarrow$	$(M, \mathrm{ISZERO}{:}\mathcal{S})$
(let)	$(\mathbf{let} < x, y >= N\ \mathbf{in}\ M, \mathcal{S})$	$\Rightarrow$	$(N, \mathbf{LET}(x,y,M){:}\mathcal{S})$
(pair1)	$(< N_1, N_2 >, \mathbf{LET}(x,y,M){:}\mathcal{S})$	$\Rightarrow$	$(M[N_1/x][N_2/y], \mathcal{S})$
(rec)	$(\mathbf{rec}\ N\ U\ V\ W, \mathcal{S})$	$\Rightarrow$	$(N, \mathbf{REC}(\mathrm{U,V,W}){:}\mathcal{S})$
(pair2)	$(< N_1, N_2 >, \mathbf{REC}(U,V,W){:}\mathcal{S})$	$\Rightarrow$	$(N_1, \mathbf{REC'}(N_2,U,V,W){:}\mathcal{S})$
(zero)	$(\overline{0}, \mathbf{REC'}(T,U,V,W){:}\mathcal{S})$	$\Rightarrow$	$(U, \mathcal{S})$
(rec1)	$(\overline{n+1}, \mathbf{REC'}(T,U,V,W){:}\mathcal{S})$	$\Rightarrow$	$(V, (\mathbf{rec}(W < \overline{n}, T >)\ U\ V\ W){:}\mathcal{S})$

Table 5.4: $\mathcal{L}_{rec}^{\mathbf{nat}}$ Stack Machine

And also the addition of the functions suc, for the successor, pre, for the predecessor, and iszero. The evaluation of these terms will be done as for the **PCF** language, as one can see by Table 5.3. Note that the call-by-name evaluation for the rest of the terms, will be identical to the one in $\mathcal{L}_{rec}$.

The Stack Machine for $\mathcal{L}_{rec}^{\mathbf{nat}}$ can be seen in Table 5.4, were the new rules for the successor, predecessor and iszero were added.

Although positive results were seen in the evaluation function, the main difference between $\mathcal{L}_{rec}$ and $\mathcal{L}_{rec}^{\mathbf{nat}}$ is noticeable in the compilation function, were the most significant changes are in the successor and predecessor, were instead of a very long $\mathcal{L}_{rec}$ expression, we now have a one-sized $\mathcal{L}_{rec}^{\mathbf{nat}}$ expression, Table 5.5.

The implementation of natural numbers on $\mathcal{L}_{rec}$, started with the addition of new data structure elements to the expression data (Code 5.6).

$$
\begin{array}{lcl}
\langle tt \rangle & = & Nat\ 0 \\
\langle ff \rangle & = & Nat\ 1 \\
\langle n \rangle & = & Nat\ n \\
\langle \mathsf{succ} \rangle & = & Suc \\
\langle \mathsf{pred} \rangle & = & Pre \\
\langle \mathsf{iszero} \rangle & = & Iszer \\
\langle \mathsf{Y}_\tau \rangle & = & \lambda f.\mathsf{rec}\ \langle Nat\ 1, Nat\ 0 \rangle\ \mathcal{M}(\langle \tau \rangle)\ f\ (\lambda x.\mathsf{let}\ \langle y, z \rangle = x\ \mathsf{in}\ \langle \mathsf{S}\ y, z \rangle) \\
\langle \mathsf{cond}_\tau \rangle & = & \lambda tuv.\mathsf{rec}\ \langle t, Nat\ 0 \rangle\ u\ (\lambda x.(\mathsf{rec}\ \langle Nat\ 0, Nat\ 0 \rangle\ (\lambda x.x)\ x\ (\lambda x.x))\ v)\ (\lambda x.x) \\
\langle x \rangle & = & x \\
\langle UV \rangle & = & \langle U \rangle \langle V \rangle \\
\langle \lambda x^\tau.T \rangle & = & \begin{cases} \lambda x.[x^\tau]\langle T \rangle & \text{if } x \in \mathrm{FV}(t) \\ \lambda x.(\mathsf{rec}\ \langle Nat\ 0, Nat\ 0 \rangle\ (\lambda x.x)\ x\ (\lambda x.x))\langle T \rangle & \text{otherwise} \end{cases}
\end{array}
$$

Table 5.5: PCF compilation into $\mathcal{L}_{rec}^{\mathbf{nat}}$

Code 5.6: $\mathcal{L}_{rec}^{\mathbf{nat}}$ Data Structure

```
data Exp = ...
    | Nat Int | Suc | Pre | Iszer
      deriving (Show, Eq)
```

With these new data structures, naturals were implemented in $\mathcal{L}_{rec}$, along with the new functions Pre for the predecessor and Iszer (similar to **PCF** iszero). Note that the successor no longer has an argument and will now be used as a function, using the *application* to apply it to an argument. The rest of the terms were kept as they were in $\mathcal{L}_{rec}$.

The implementation of the call-by-name evaluation and Stack Machine can be seen in Appendix Code A.16 and Code A.17, where the code is almost a merge between the $\mathcal{L}_{rec}$ code and the **PCF** code, and therefore needs no explanation. The same goes for the compilation function (Appendix Code A.18).

In the next section we will show the results of this new implementation.

5.2.2 Comparing Results with $\mathcal{L}_{rec}^{\mathbf{nat}}$

For the results of the abstract time measure cost (Section 5.2) in the $\mathcal{L}_{rec}^{\mathbf{nat}}$ system, the evaluation function of this system was also changed (Appendix Code A.19). Note that in this code, there are also special cases for some reductions that do not calculate the cost, as it happened for the **PCF** language and System $\mathcal{L}_{rec}$.

The results for this new function can be seen on Table 5.6, where the values of the

	Abstract Time Measure			Number of reductions		
Function	**PCF**	$\mathcal{L}_{rec}^{\mathbf{nat}}$	**PCF to** $\mathcal{L}_{rec}^{\mathbf{nat}}$	**PCF**	$\mathcal{L}_{rec}^{\mathbf{nat}}$	**PCF to** $\mathcal{L}_{rec}^{\mathbf{nat}}$
add 2 3	56	31	871	23	11	135
mult 2 3	247	105	10703	87	37	486
fib 4	1255	434	45202	197	130	1091
fact 4	22165	3717	98520	4546	993	22335

Table 5.6: Results Comparison with $\mathcal{L}_{rec}^{\mathbf{nat}}$

	Abstract Time Measure		Number of reductions	
Function	**PCF to** $\mathcal{L}_{rec}$	**PCF to** $\mathcal{L}_{rec}^{\mathbf{nat}}$	**PCF to** $\mathcal{L}_{rec}$	**PCF to** $\mathcal{L}_{rec}^{\mathbf{nat}}$
add 2 3	4975	871	503	135
mult 2 3	5091	10703	3012	486
fib 4	289499	45202	18356	1091
fact 4	5464324	98520	345722	22335

Table 5.7: Results Comparison Between Compilations

cost and number of reductions for $\mathcal{L}_{rec}^{\mathbf{nat}}$ are better than $\mathcal{L}_{rec}$ for small functions, as add and mult, but get higher with more complex functions. The results for the number of reductions show that $\mathcal{L}_{rec}$ and $\mathcal{L}_{rec}^{\mathbf{nat}}$ have smaller number of reductions needed to evaluate an expression than the **PCF** language, even though the evaluation with the encoding takes a lot more reductions to achieve normal form. This is expected since the compilation from **PCF** to $\mathcal{L}_{rec}$ turns the expression terms of **PCF** into term in $\mathcal{L}_{rec}$ of a much larger length, and therefore will have more reductions before achieving normal form. Note that the number of reductions is higher in the encoding from **PCF** to $\mathcal{L}_{rec}$, than that of the encoding from **PCF** to $\mathcal{L}_{rec}^{\mathbf{nat}}$ (Table 5.7), which is also normal since $\mathcal{L}_{rec}^{\mathbf{nat}}$ has smaller length terms than $\mathcal{L}_{rec}$ after encoding them from **PCF**. However, note that the number of reductions is smaller in $\mathcal{L}_{rec}$ than that of $\mathcal{L}_{rec}^{\mathbf{nat}}$, since terms originally created in $\mathcal{L}_{rec}$ are faster to evaluate than that of terms originally created in $\mathcal{L}_{rec}^{\mathbf{nat}}$.

As for the abstract time measure, as one can see in Figure 5.1, **PCF** terms have a much higher cost than the terms of $\mathcal{L}_{rec}$ and $\mathcal{L}_{rec}^{\mathbf{nat}}$, specially in more complex functions. Which comes to show that System $\mathcal{L}_{rec}$ is truly more efficient evaluating expressions with a recursor, than the fixed point operator of the **PCF**.

In Figure 5.2, one can see the difference between encoding from **PCF** to $\mathcal{L}_{rec}$ and

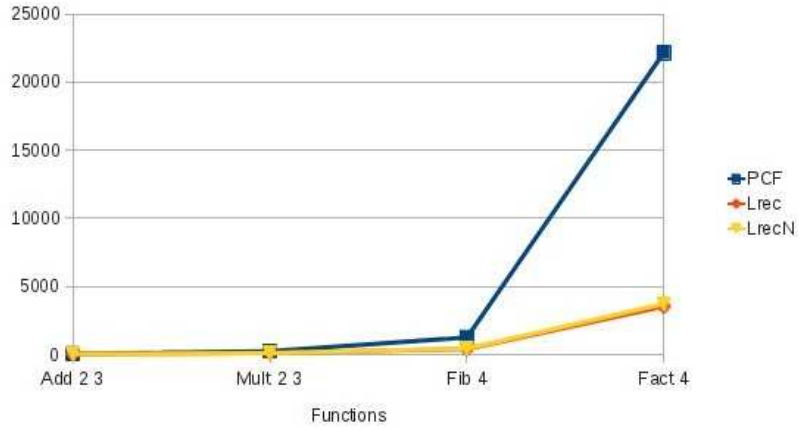

Figure 5.1: Abstract time measure results

encoding from **PCF** to $\mathcal{L}_{rec}^{\mathbf{nat}}$. Once again, this happens because when the terms are encoded to $\mathcal{L}_{rec}$, their length increases almost exponentially which makes them more expensive to reduce to normal form. And when the terms are encoded to $\mathcal{L}_{rec}^{\mathbf{nat}}$, due to the built-in naturals, the succ, the pred and the iszero function, the terms length do not have such a large increase and therefore do not have to go through so many reductions in order to achieve normal form. The main reason relies on the problem that if the function has a large natural number, $\mathcal{L}_{rec}$ will transform it into a long line of application of successors that complicates the evaluation.
In the next chapter we will take some conclusions about these results.

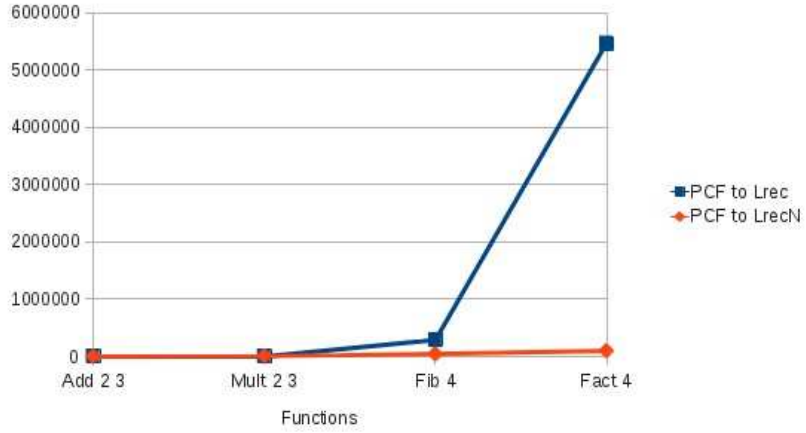

Figure 5.2: Abstract time measure results for the encodings

Chapter 6

Conclusion

Throughout this thesis we discussed the implementation of:

- **PCF**, a λ-calculus based language;
- System $\mathcal{L}_{rec}$, a linear language, based on the linear λ-calculus;
- $\mathcal{L}_{rec}^{\mathbf{nat}}$, an extension of System $\mathcal{L}_{rec}$ with built-in naturals.

The implementations were started with a parser in Happy that encoded the input in **PCF**, $\mathcal{L}_{rec}$ or $\mathcal{L}_{rec}^{\mathbf{nat}}$, into a created data structure in Haskell, in order to evaluate the expressions of the languages. We also compare the abstract time measure of their evaluation in order to assess the efficiency of $\mathcal{L}_{rec}$ when compared to **PCF**.
For some of the functions analyzed, the encoding in $\mathcal{L}_{rec}$ proved to be more efficient than the respective encoding in **PCF** and even more than the encoding resulting from translating from **PCF** to $\mathcal{L}_{rec}$. In that sense, future work can be done in the encoding function from **PCF** to $\mathcal{L}_{rec}$, so that the length of terms do not increase so drastically and therefore improve the evaluation of such terms.
This improvement would have to take advantage of the recursor. Note that in functional programming languages a way to improve poor performance programs is to transform recursive functions into more efficient versions [Harrison 92], sometimes using an iterative solution. This can be a technique used when translating programs written using Y into programs using the $\mathcal{L}_{rec}$ recursor.
Note also that with the built-in naturals, System $\mathcal{L}_{rec}$ showed improved results when encoding from **PCF** to $\mathcal{L}_{rec}^{\mathbf{nat}}$, which supports that $\mathcal{L}_{rec}$ can also be improved with built-in arithmetic functions and by adding constructors so it can be used as a pro-

gramming language or an intermediate code, as the Spineless Tagless G-machine (STG [Jones 92]) is for Haskell.

Appendix A

Code

A.1 PCF Language

Code A.1: Evaluation Function

```
eval :: ExpPCF -> ExpPCF
eval (Var a)      = (Var a)
eval (Lambd x m) = (Lambd x m)
eval (Nat x)      = (Nat x)
eval (Succ)       = (Succ)
eval (Pred)       = (Pred)
eval (Cond)       = (Cond)
eval (IsZ)        = (IsZ)
eval (PtoF )      = (PtoF)
eval (App PtoF m)                    = eval(App m (App PtoF m))
eval (App (Lambd x m) n)             = eval(substt m x n)
eval (App (App (App Cond TT) m) n) = eval (m)
eval (App (App (App Cond FF) m) n) = eval (n)
eval (App (App (App Cond b) m) n)  =
                eval (App (App (App Cond (eval b)) m) n)
eval (App Succ m)                    = case (eval m) of
           (Nat n)   -> (Nat (n+1))
           t         -> (App Succ t)
eval (App Pred m)                    = case (eval m) of
            (Nat (n+1)) -> (Nat n)
            t           -> (App Pred t)
eval (App IsZ (Nat n))               = if (n==0) then (TT) else (FF)
eval (App IsZ m)                     = eval(App IsZ (eval m))
eval (App s t)                       = eval_aux(App (eval s) t)
eval t                               = t
```

Code A.2: Substitution Function

```
substt :: ExpPCF -> String -> ExpPCF -> ExpPCF
substt (Nat a) y l = (Nat a)
substt (Succ)  y l = (Succ)
substt (Pred)  y l = (Pred)
substt (IsZ)   y l = (IsZ )
substt (PtoF ) y l = (PtoF)
substt (Cond)  y l = (Cond)
substt (Var x) y l
  | x == y            = l
  | otherwise         = (Var x)
substt (Lambd x m ) y l
  | x==y              = (Lambd x m)
  | otherwise         = (Lambd x (substt m y l))
substt (App m n)  y l = (App (substt m y l) (substt n y l)
```

Code A.3: Stack Function

```
stck :: (ExpPCF, Stck) -> (ExpPCF, Stck)
stck ((App m n), s)                  = stck (m,(E n):s)
stck ((Lambd x m), (E n):s)          = stck ((substt m x n),s)
stck ((PtoF), (E m):s)               = stck (m, (E (App PtoF m)):s)
stck (Cond, (E m):(E n1):(E n2):s)   = stck (m, COND(n1,n2):s)
stck (TT, (COND (n1,n2)):s)          = stck (n1,s)
stck (FF, (COND (n1,n2)):s)          = stck (n2,s)
stck ((Nat n), (E (Succ)):s)         = stck ((Nat (n+1)), s)
stck ((Succ), (E m):s)               = stck (m,(E (Succ)):s)
stck ((Nat 0), (E (Pred)):s)         = stck ((Nat 0), s)
stck ((Nat (n+1)), (E (Pred)):s)     = stck ((Nat n), s)
stck ((Pred), (E m):s)               = stck (m,(E (Pred)):s)
stck ((Nat 0),(E (IsZ)):s)           = stck (TT,s)
stck ((Nat (n+1)),(E (IsZ)):s)       = stck (FF,s)
stck ((IsZ ), (E m):s)               = stck (m, (E (IsZ)):s)
stck (v,s)                           = (v,s)
```

A.2 $\mathcal{L}_{rec}$ Language

Code A.4: call-by-name Evaluation Function

```
eval_cbn :: Exp -> Exp
eval_cbn (Suc a)
```

```
  | (value a)          = (Suc a)
  | otherwise          = (Suc (eval_cbn a))
eval_cbn (Lambd a b) = (Lambd a b)
eval_cbn (Pair a b)  = (Pair a b)
eval_cbn (App (Lambd x u) t) = eval_cbn(substt u x t)
eval_cbn (App s t) = cbn_aux(App (eval_cbn s) t)
eval_cbn (Let (x, y) (Pair t1 t2) u) =
              eval_cbn (App (App (Lambd x (Lambd y (u))) t1) t2)
eval_cbn (Let (a, b) p  u) =
              eval_cbn (Let (a, b) (eval_cbn p) u)
eval_cbn (Rec (Pair (Zero) t2) u v w) = eval_cbn u
eval_cbn (Rec (Pair (Suc t) t2) u v w) =
              eval_cbn (App v (Rec (App w (Pair t t2)) u v w))
eval_cbn (Rec (Pair p_1 p_2) u v w) =
        eval_cbn (Rec (Pair (eval_cbn p_1) p_2) u v w)
eval_cbn (Rec p u v w) = eval_cbn (Rec (eval_cbn p) u v w)
eval_cbn t                                  = t
```

Code A.5: call-by-value Evaluation Function

```
eval_cbv :: Exp -> Exp
eval_cbv (Zero) = (Zero)
eval_cbv (Var a) = (Var a)
eval_cbv (Suc a)
  | (value a)          = (Suc a)
  | otherwise          = (Suc (eval_cbv a))
eval_cbv (Lambd a b) = (Lambd a b)
eval_cbv (Pair a b)  = (Pair a b)
eval_cbv (App (Lambd x u) t) = eval_cbv(substt u x (eval_cbv t))
eval_cbv (App s           t) = cbv_aux(App (eval_cbv s) t)
eval_cbv (Let (x,y) (Pair t1 t2) u)  =
              eval_cbv (App (App (Lambd x (Lambd y (u))) t1) t2)
eval_cbv (Rec (Pair (Zero) t2) u v w) = eval_cbv u
eval_cbv (Rec (Pair (Suc t)   t2) u v w) =
        eval_cbv (App v (Rec (App w (Pair t t2)) u v w))
eval_cbv (Rec  p u v w) = eval_cbv (Rec (eval_cbn p) u v w)
eval_cbv t                                      = t
```

Code A.6: Auxiliary Evaluation Function for CBV

```
cbv_aux :: Exp -> Exp
cbv_aux (App (Lambd x u) t) = eval_cbv(substt u x (eval_cbv t))
cbv_aux (App s     t) = (App s t)
```

Code A.7: Substitution Function

```
substt :: Exp -> String -> Exp -> Exp
substt (Zero) y l = (Zero )
substt (Var x) y l
  | x == y              = l
  | otherwise           = (Var x)
substt (Lambd x m ) y l
  | x==y                = (Lambd x m)
  | otherwise           = (Lambd x (substt m y l))
substt (App m n)  y l = (App (substt m y l) (substt n y l))
substt (Pair m n) y l = (Pair (substt m y l) (substt n y l))
substt (Suc x)    y l = (Suc (substt x y l))
substt (Let (x,z) p u) y l
  | x==y || z==y      = (Let (x,z) p u)
  | otherwise       = (Let (x,z) (substt p y l) (substt u y l))
substt (Rec t u v w) y l =
       (Rec (substt t y l) (substt u y l) (substt v y l) (substt w y l))
```

Code A.8: Stack Function

```
stck :: (Exp, Stck) -> (Exp, Stck)
stck ((App m n),s)                    = stck (m,(E n):s)
stck ((Lambd x m),(E n):s)            = stck ((substt m x n),s)
stck ((Let (x,y) n m),s)              = stck (n, LET(x,y,m):s)
stck ((Pair n1 n2), (LET (x,y,m)):s) =
      stck ((substt (substt m x n1) y n2), s)
stck ((Rec n u v w), s)               = stck (n, REC(u,v,w):s)
stck ((Pair n1 n2), (REC (u,v,w)):s) = stck (n1, RECC(n2, u, v, w):s)
stck ((Zero), (RECC (t,u,v,w)):s)     = stck (u,s)
stck ((Suc n), (RECC (t,u,v,w)):s)    =
      stck (v,(E (Rec (App w (Pair n t)) u v w)):s)
stck ((Suc n), SUCC:s)                =
      if (value n) then stck (Suc (Suc n), s) else stck (n, SUCC:SUCC:s)
stck ((Suc n), s)                     =
      if (value n) then ((Suc n),s) else stck (n, SUCC:s)
stck (n, SUCC:s)                      =
      if (value n) then stck ((Suc n),s) else ((Suc n),s)
stck ( v,s)                           = (v,s)
```

A.3 Compilation

Code A.9: Compilation Function

```
comp :: ExpPCF -> Exp
```

```
comp (TT)                  = (Zero )
comp (FF)                  = (Suc (Zero ))
comp (NatPCF 0)            = (Zero )
comp (NatPCF (n+1))        = (Suc (comp_nat n))
comp (Succ)                =
    (Lambd "n" (Rec (Pair (Var "n") (Zero )) (Suc (Zero ))
                    (Lambd "x" (Suc (Var "x"))) idntt))
comp (Pred)                =
  (Lambd "n" (App pr1 (Rec (Pair (Var "n") (Zero ))(Pair (Zero)(Zero))
     (Lambd "x" (Let ("t","u") (App (dup (TInt)) (App pr2 (Var "x")))
       (Pair (Var "t") (Suc (Var "u"))))) idntt )))
comp (IsZ)                 =
  (Lambd "n" (App pr1 (Rec (Pair (Var "n") (Zero ))
     (Pair (Zero) (Suc (Zero))) (Lambd "x" (App (dup (TInt))
       (App pr2 (Var "x")))) idntt)))
comp (PtoF a)              =
  (Lambd "f" (Rec (Pair (Suc (Zero)) (Zero)) (make a) (Var "f")
     (Lambd "x" (Let ("y","z") (Var "x")
       (Pair (Suc (Var "y")) (Var "z"))))))
comp (Cond)                =
  (Lambd "t" (Lambd "u" (Lambd "v" (Rec (Pair (Var "t") (Zero ))
     (Var "u") (Lambd "x" (App (Rec (Pair (Zero)(Zero)) idntt
       (Var "x") idntt) (Var "v"))) idntt))))
comp (VarPCF x)            = (Var x)
comp (AppPCF u v)          = (App (comp u) (comp v))
comp (LambdPCF (x,tp) t)
 | (elem (VarPCF x) (fvcPCF t))=(Lambd x (linearFV (Var x, tp)(comp t)))
 | otherwise =
  (Lambd x (App (Rec (Pair (Zero)(Zero)) idntt (Var x) idntt)(comp t)))
```

Code A.10: Linearization Function

```
linearFV :: (Exp,Type) -> Exp -> Exp
linearFV (x,tp) (Suc u)      = (Suc (linearFV (x,tp) u))
linearFV (x,tp) (Lambd y u) = (Lambd y (linearFV (x,tp) u))
linearFV (x,tp) (App s u)
  | (elem x (fvc s)) && (elem x (fvc u))  =
     (copy ("x1","x2") (x,tp) (App (substt (linearFV (x,tp) s) "x1" x)
                      (substt (linearFV (x,tp) u) "x2" x)))
  | not (elem x (fvc u))          = (App (linearFV (x,tp) s) u)
  | not (elem x (fvc s))          = (App s (linearFV (x,tp) u))
linearFV (x,tp) (Let (a,b) u v)   = (Let (a,b) u (linearFV (x,tp) v))
linearFV (x,tp) (Var y)           = x
```

Code A.11: Erase Function

```
erase :: (Exp,Type) -> Exp
erase (t,TInt)      = (Rec (Pair t (Zero )) idntt idntt idntt)
erase (t,TBool)     = (Rec (Pair t (Zero )) idntt idntt idntt)
erase (t,TApp ta tb) = erase ((App t (make ta)),tb)
```

Code A.12: Make Function

```
make :: Type -> Exp
make (TInt)       = (Zero )
make (TBool)      = (Zero )
make (TApp ta tb) = (Lambd "x" (App (erase ((Var "x"),ta)) (make tb)))
```

Code A.13: **PCF** expression size

```
esizePCF :: ExpPCF -> Int
esizePCF (TT) = 1
esizePCF (FF) = 1
esizePCF (NatPCF n) = 1
esizePCF (Succ) = 1
esizePCF (Pred) = 1
esizePCF (IsZ) = 1
esizePCF (PtoF tp) = 1
esizePCF (Cond) = 1
esizePCF (VarPCF x) = 1
esizePCF (AppPCF n m) = (esizePCF n)+(esizePCF m)
esizePCF (LambdPCF (x,tp) t) = 1+(esizePCF t)
```

Code A.14: $\mathcal{L}_{rec}$ evaluation with Abstract Time Measure

```
type Counter = Int
type Cost = Int

eval_cbn :: Exp -> Counter -> (Exp, Counter, Cost)
eval_cbn (Lambd x u) n = ((Lambd x u),n,0)
eval_cbn (Pair a b)  n = ((Pair a b),n,0)
eval_cbn (Suc a) n
  | (value a)         = ((Suc a),n,0)
  | otherwise         =
        let (e,nr,it)=(eval_cbn a (n+1))
            fi = (max 1 ((esizeLRec a)-(esizeLRec (Suc a))))
        in ((Suc e),nr,fi+it)
eval_cbn (App (Lambd x u) t) n =
        let (e,nr,it) = (eval_cbn (substt u x t) (n+1))
            fi = (max 1 ((esizeLRec (substt u x t))-
                         (esizeLRec (App (Lambd x u) t))))
        in (e,nr,fi+it)
```

```
eval_cbn (App s           t) n =
         let (e1,nr1,it) = (eval_cbn s n)
             (e2,nr2,it2) = (cbn_aux(App e1 t) nr1)
         in (e2, nr2,it2+it)
eval_cbn (Let (x, y) (Pair t1 t2) u) n =
        let (e,nr,it) = (eval_cbn (substt (substt u x t1) y t2) (n+1))
            fi = (max 1 ((esizeLRec (substt (substt u x t1) y t2))-
                         (esizeLRec (Let (x, y) (Pair t1 t2) u))))
        in (e,nr,fi+it)
eval_cbn (Let (a, b)  p           u) n =
         let (e2,nr2,it) = (eval_cbn p n)
             (e1,nr1,it2) = (eval_cbn (Let (a, b) e2 u) nr2)
         in (e1,nr1,it2+it)
eval_cbn (Rec (Pair (Zero) t2) u v w)    n =
         let (e,nr,it)= (eval_cbn u (n+1))
             fi = (max 1 ((esizeLRec u)-
                          (esizeLRec (Rec (Pair (Zero) t2) u v w))))
         in (e,nr,fi+it)
eval_cbn (Rec (Pair (Suc t)   t2) u v w) n =
     let (e,nr,it)=(eval_cbn(App v (Rec(App w(Pair t t2)) u v w)) (n+1))
         fi = (max 1 ((esizeLRec(App v (Rec(App w(Pair t t2)) u v w)))-
                      (esizeLRec (Rec (Pair (Suc t) t2) u v w))))
     in (e,nr,fi+it)
eval_cbn (Rec (Pair p_1 p_2)      u v w) n =
     let (e2,nr2,it) = (eval_cbn p_1 n)
         (e1,nr1,it2) = (eval_cbn (Rec (Pair e2 p_2) u v w) nr2)
     in (e1,nr1,it2+it)
eval_cbn (Rec p                   u v w) n =
     let (e2,nr2,it) = (eval_cbn p n)
         (e1,nr1,it2) = (eval_cbn (Rec e2 u v w) nr2)
     in (e1,nr1,it2+it)
eval_cbn t n = (t,n,0)
```

Code A.15: **PCF** evaluation with Abstract Time Measure

```
type Counter = Int
type Cost = Int

eval :: ExpPCF -> Counter -> (ExpPCF,Counter,Cost)
eval (Var a)      n = ((Var a),n,0)
eval (Lambd x m) n = ((Lambd x m),n,0)
eval (Nat x)      n = ((Nat x),n,0)
eval (Succ)       n = ((Succ),n,0)
eval (Pred)       n = ((Pred),n,0)
eval (Cond)       n = ((Cond),n,0)
```

```
eval (IsZ)        n = ((IsZ),n,0)
eval (PtoF )      n = ((PtoF),n,0)
eval (App PtoF m) n = let (e,nr,c) = (eval(App m (App PtoF m)) (n+1))
                          fc = (esizePCF m)
                      in (e,nr,fc+c)
eval (App (Lambd x m) n) nr =
   let (e,nr1,c) = (eval (substt m x n) (nr+1))
       fc = (max 1 ((esizePCF (substt m x n))-
                    (esizePCF (App (Lambd x m) n))))
   in (e,nr1,fc+c)
eval (App (App (App Cond TT) m) n) nr =
   let (e,nr1,c)=(eval m (nr+1))
       fc = (max 1 ((esizePCF m)-
                    (esizePCF (App (App (App Cond TT) m) n))))
   in (e,nr1,c+fc)
eval (App (App (App Cond FF) m) n) nr =
   let (e,nr2,c) = (eval n (nr+1))
       fc = (max 1 ((esizePCF n)-
                    (esizePCF (App (App (App Cond FF) m) n))))
   in (e,nr2,c+fc)
eval (App (App (App Cond b) m) n)  nr =
   let (e1,nr1,c1) = (eval b nr)
       (e2,nr2,c2) = (eval (App (App (App Cond e1) m) n) nr1)
   in (e2,nr2,c2+c1)
eval (App Succ m) n = case (eval m n) of
                          ((Nat n),nr,c)   -> ((Nat (n+1)),(nr+1),c+1)
                          (t,nr,c)         -> ((App Succ t),nr,c)
eval (App Pred m) n = case (eval m n) of
                  ((Nat (n+1)),nr,c) -> ((Nat n),(nr+1),c+1)
                  (t,nr,c)           -> ((App Pred t),nr,c)
eval (App IsZ (Nat n)) nr = if (n==0)
                            then (TT,(nr+1),1)
                            else (FF,(nr+1),1)
eval (App IsZ m) n = let (e1,nr1,c) = (eval m n)
                         (e2,nr2,c2) = (eval(App IsZ e1) nr1)
                     in (e2,nr2,c+c2)
eval (App s t) n = let (e1,nr1,c) = (eval s n)
                       (e2,nr2,c2) = (eval_aux(App e1 t) nr1)
                   in (e2,nr2,c+c2)
eval t n = (t,n,0)
```

A.3.1 $\mathcal{L}_{rec}^{\mathbf{nat}}$ Code

Code A.16: $\mathcal{L}_{rec}^{\mathbf{nat}}$ call-by-name Evaluation Function

```
eval_cbn :: Exp -> Exp
eval_cbn (Lambd a b) = (Lambd a b)
eval_cbn (Pair a b)  = (Pair a b)
eval_cbn (App (Lambd x u) t) = eval_cbn(substt u x t)
eval_cbn (App (Iszer)     n) = if ((eval_cbn n) == (Nat 0)) then (Nat 0)
    else (Nat 1)
eval_cbn (App (Suc)       n) = if (isnumber num) then (Nat (numS num))
   else (App (Suc) n)
    where
      num = (eval_cbn n)
eval_cbn (App (Pre)       n) = if (isnumber num) then (Nat (numP num))
   else (App (Pre) n)
    where
      num = (eval_cbn n)
eval_cbn (App s           t) = cbn_aux(App (eval_cbn s) t)
eval_cbn (Let (x, y) (Pair t1 t2) u) = eval_cbn (App (App (Lambd x (
   Lambd y (u))) t1) t2)
eval_cbn (Let (a, b)  p           u) = eval_cbn (Let (a, b) (eval_cbn p)
    u)
eval_cbn (Rec (Pair (Nat 0) t2) u v w)       = eval_cbn u
eval_cbn (Rec (Pair (Nat (n+1))   t2) u v w) = eval_cbn (App v (Rec (App
    w (Pair (Nat n) t2)) u v w))
eval_cbn (Rec (Pair p_1 p_2)      u v w)     = eval_cbn (Rec (Pair (
   eval_cbn p_1) p_2) u v w)
eval_cbn (Rec p                   u v w)     = eval_cbn (Rec (eval_cbn p
   ) u v w)
eval_cbn t                                   = t
```

Code A.17: $\mathcal{L}_{rec}^{\mathbf{nat}}$ Stack Machine

```
stck :: (Exp, Stck) -> (Exp, Stck)
stck ((App m n),s)                  = stck (m,(E n):s)
stck ((Lambd x m),(E n):s)          = stck ((substt m x n),s)
stck ((Nat n), (E (Suc)):s)         = stck ((Nat (n+1)),s)
stck ((Suc), (E m):s)               = stck (m, (E (Suc)):s)
stck ((Nat 0), (E (Pre)):s)         = stck ((Nat 0), s)
stck ((Nat (n+1)), (E (Pre)):s)     = stck ((Nat n), s)
stck ((Pre), (E m):s)               = stck (m,(E (Pre)):s)
stck ((Nat 0), (E Iszer):s)         = stck ((Nat 0),s) -- True
stck ((Nat (n+1)), (E Iszer):s)     = stck ((Nat 1),s) -- False
stck ((Iszer), (E n):s)             = stck (n, (E Iszer):s)
stck ((Let (x,y) n m),s)            = stck (n, LET(x,y,m):s)
stck ((Pair n1 n2), (LET (x,y,m)):s) =
              stck ((substt (substt m x n1) y n2), s)
```

```
stck ((Rec n u v w), s)                 = stck (n, REC(u,v,w):s)
stck ((Pair n1 n2), (REC (u,v,w)):s) = stck (n1, RECC(n2, u, v, w):s)
stck ((Nat 0), (RECC (t,u,v,w)):s)    = stck (u,s)
stck ((Nat n), (RECC (t,u,v,w)):s)    =
                stck (v,(E (Rec (App w (Pair (Nat (n-1)) t)) u v w)):s)
stck ( v,s)                             = (v,s)
```

Code A.18: **PCF** to $\mathcal{L}_{rec}^{\mathbf{nat}}$ compilation

```
comp :: ExpPCF -> Exp
comp (TT)        = (Nat 0 )
comp (FF)        = (Nat 1 )
comp (NatPCF n) = (Nat n )
comp (Succ)      = (Suc)
comp (Pred)      = (Pre)
comp (IsZ)       = (Iszer)
comp (PtoF a) = (Lambd "f" (Rec (Pair (Nat 1 ) (Nat 0 ))
                      (make a) (Var "f") (Lambd "x"
                           (Let ("y","z") (Var "x")
                              (Pair (App (Suc) (Var "y")) (Var "z"))))))
comp (Cond) = (Lambd "t" (Lambd "u" (Lambd "v"
                  (Rec (Pair (Var "t") (Nat 0 )) (Var "u")
                       (Lambd "x" (App (Rec
                           (Pair (Nat 0) (Nat 0 )) idntt (Var "x") idntt)
                              (Var "v"))) idntt))))
comp (VarPCF x)           = (Var x)
comp (AppPCF u v)         = (App (comp u) (comp v))
comp (LambdPCF (x,tp) t)
  | (elem (VarPCF x) (fvcPCF t)) =
           (Lambd x (linearFV  (Var x, tp) (comp t)))
  | otherwise =
           (Lambd x (App (Rec (Pair (Nat 0 ) (Nat 0 )) idntt (Var x)
              idntt ) (comp t)))
```

Code A.19: $\mathcal{L}_{rec}^{\mathbf{nat}}$ with Abstract Time Measure

```
eval_cbn :: Exp -> Counter -> (Exp, Counter, Cost)
eval_cbn (Lambd x u) n = ((Lambd x u),n,0)
eval_cbn (Pair a b)  n = ((Pair a b),n,0)
eval_cbn (App (Lambd x u) t) n =
  let (e,nr,it) = (eval_cbn (substt u x t) (n+1))
      fi = (max 1 ((esizeLRecN (substt u x t))-
                   (esizeLRecN (App (Lambd x u) t))))
  in (e,nr,fi+it)
eval_cbn (App (Iszer) (Nat n)) nr = if (n==0)
                                    then ((Nat 0),(nr+1),1)
```

```
                                    else ((Nat 1),(nr+1),1)
eval_cbn (App (Iszer)      m) n =
  let (e1,nr1,c) = (eval_cbn m n)
      (e2,nr2,c2) = (eval_cbn (App Iszer e1) nr1)
  in (e2,nr2,c+c2)
eval_cbn (App (Suc)        m)  n =
  let (e,nr,c) = (eval_cbn m n)
      c1 = (max 1 ((esizeLRecN (App Suc e)) - (esizeLRecN (App Suc m))))
  in if (isnumber e)
       then ((Nat (numS e)),(nr+1),c+1) else ((App (Suc) e),nr,c+c1)
eval_cbn (App Pre        m) n =
  let (e,nr,c) = (eval_cbn m n)
      c1 = (max 1 ((esizeLRecN m)-(esizeLRecN (App Pre m))))
  in if (isnumber e)
       then ((Nat (numP e)),(nr+1),c+1) else ((App (Pre) m),nr,c+c1)
eval_cbn (App s            t) n =
    let (e1,nr1,c) = (eval_cbn s n)
        (e2,nr2,c2) = (cbn_aux (App e1 t) nr1)
    in (e2,nr2,c+c2)
eval_cbn (Let (x, y) (Pair t1 t2) u) n =
  let (e,nr,c) = (eval_cbn (substt (substt u x t1) y t2) (n+1))
      fc = (max 1 ((esizeLRecN (substt (substt u x t1) y t2))-
                   (esizeLRecN (Let (x, y) (Pair t1 t2) u))))
  in (e,nr,fc+c)
eval_cbn (Let (a, b)  p            u) n =
  let (e,nr,c) = (eval_cbn p n)
      (e1,nr1,c1) = (eval_cbn (Let (a, b)  e            u) nr)
  in (e1,nr1,c+c1)
eval_cbn (Rec (Pair (Nat 0) t2) u v w)  n =
  let (e,nr,it) = (eval_cbn u (n+1))
      fc = it+(max 1 ((esizeLRecN u)-
                      (esizeLRecN (Rec (Pair (Nat 0) t2) u v w))))
  in (e,nr,fc)
eval_cbn (Rec (Pair (Nat (n+1))   t2) u v w) nr =
 let (e,nr1,it) =
       (eval_cbn (App v (Rec (App w (Pair (Nat n) t2)) u v w)) (nr+1))
     fc = it + (max 1
            ((esizeLRecN (App v (Rec (App w (Pair (Nat n) t2)) u v w)))-
             (esizeLRecN (Rec (Pair (Nat (n+1))   t2) u v w))))
  in (e,nr1,fc)
eval_cbn (Rec (Pair p_1 p_2)       u v w)  n  =
    let (e2,nr2,it) = (eval_cbn p_1 n)
        (e1,nr1,it2) = (eval_cbn (Rec (Pair e2 p_2) u v w) nr2)
    in (e1,nr1,it2+it)
eval_cbn (Rec p                    u v w)  n  =
```

```
    let (e2,nr2,it) = (eval_cbn p n)
57      (e1,nr1,it2) = (eval_cbn (Rec e2 u v w) nr2)
    in (e1,nr1,it2+it)
59 eval_cbn t       n                                = (t,n,0)
```

References

[Alves 10] Sandra Alves, Maribel Fernández, Mário Florido & Ian Mackie. *Gödel's system T revisited*. Theor. Comput. Sci., vol. 411, no. 11-13, pages 1484–1500, March 2010.

[Alves 11] Sandra Alves, Maribel Fernández, Mário Florido & Ian Mackie. *Linearity and recursion in a typed Lambda-calculus*. In Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming, PPDP '11, pages 173–182, New York, NY, USA, 2011. ACM.

[Asperti 98] Andrea Asperti. *Light Affine Logic*. In LICS, pages 300–308, 1998.

[Asperti 02] Andrea Asperti & Luca Roversi. *Intuitionistic Light Affine Logic*. ACM Trans. Comput. Logic, vol. 3, no. 1, pages 137–175, January 2002.

[Baillot 04] Patrick Baillot & Virgile Mogbil. *Soft lambda-Calculus: A Language for Polynomial Time Computation*. In FoSSaCS, pages 27–41, 2004.

[Barendregt 84] Henk Barendregt. The lambda calculus: Its syntax and semantics, volume 2. North-Holland, 1984.

[Barendregt 97] Henk Barendregt. *The Impact of the Lambda Calculus on Logic and Computer Science*. Bulletin of Symbolic Logic, vol. 3, no. 3, pages 181–215, 1997.

[Church 32] Alonzo Church. *A Set of Postulates for the Foundation of Logic*. Annals of Mathematics, vol. 33, no. 2, pages 346–366, 1932.

[Church 36] A. Church & J. B. Rosser. *Some Properties of Conversion*. Transactions of the American Mathematical Society, vol. 39, no. 3, pages 472–482, 1936.

[Church 40] Alonzo Church. *A Formulation of the Simple Theory of Types*. The Journal of Symbolic Logic, vol. 5, no. 2, pages 56–68, 1940.

[Curry 34] H. B. Curry. *Functionality in Combinatory Logic*. In Proceedings of the National Academy of Sciences of the United States of America, pages 584–590, November 1934.

[Curry 58] Haskell B. Curry & R. Feys. Combinatory logic, volume 1. North Holland, 1958.

[Gill 01] Andy Gill & Simon Marlow. *Happy: The Parser Generator for Haskell*, September 2001.

[Girard 87] Jean-Yves Girard. *Linear Logic*. Theor. Comput. Sci., vol. 50, pages 1–102, 1987.

[Girard 89] Jean Y. Girard, Paul Taylor & Yves Lafont. Proofs and types. Cambridge University Press, New York, NY, USA, 1989.

[Girard 98] Jean-Yves Girard. *Light Linear Logic*. Inf. Comput., vol. 143, no. 2, pages 175–204, 1998.

[Harrison 92] Peter G. Harrison & Hessam Khoshnevisan. *A New Approach to Recursion Removal*. Theor. Comput. Sci., vol. 93, no. 1, pages 91–113, 1992.

[Has 03] Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, May 2003.

[Hofmann 99] Martin Hofmann. *Linear Types and Non Size-Increasing Polynomial Time Computation*. Logic in Computer Science, Symposium on, vol. 0, page 464, 1999.

[Jones 92] Simon L. Peyton Jones. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5*. Journal of Functional Programming, vol. 2, pages 127–202, 1992.

[Krivine 07] Jean-Louis Krivine. *A call-by-name lambda-calculus machine*. Higher Order Symbol. Comput., vol. 20, no. 3, pages 199–207, September 2007.

[Lafont 04] Yves Lafont. *Soft linear logic and polynomial time*. Theor. Comput. Sci., vol. 318, no. 1-2, pages 163–180, June 2004.

[Lago 06] Ugo Dal Lago & Simone Martini. *An Invariant Cost Model for the Lambda Calculus*. In CiE, pages 105–114, 2006.

[Lago 09] Ugo Dal Lago. *The geometry of linear higher-order recursion*. ACM Trans. Comput. Log., vol. 10, no. 2, 2009.

[Landin 64] P. J. Landin. *The Mechanical Evaluation of Expressions*. The Computer Journal, vol. 6, no. 4, pages 308–320, January 1964.

[Plotkin 77] Gordon D. Plotkin. *LCF Considered as a Programming Language*. Theor. Comput. Sci., vol. 5, no. 3, pages 223–255, 1977.

[Scott 93] Dana S. Scott. *A type-theoretical alternative to ISWIM, CUCH, OWHY*. Theoretical Computer Science, vol. 121, no. 1â2, pages 411 – 440, 1993.

[Sestoft 02] Peter Sestoft. *The essence of computation*. chapitre Demonstrating lambda calculus reduction, pages 420–435. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Terui 01] Kazushige Terui. *Light Affine Calculus and Polytime Strong Normalization*. In LICS, pages 209–220, 2001.